FEDERAL RURAL UNIVERSITY OF PERNAMBUCO

POSTGRADUATE PROGRAM IN APPLIED INFORMATICS

SAULO LUCAS GOMES FERREIRA

PERFORMANCE IMPACTS OF NOSQL DATABASES CONSISTENCY LEVELS IN MULTIPLE ENVIRONMENTS

RECIFE – PE

2025

SAULO LUCAS GOMES FERREIRA

PERFORMANCE IMPACTS OF NOSQL DATABASES CONSISTENCY LEVELS IN MULTIPLE ENVIRONMENTS

Master dissertation presented to the Postgraduate Program in Applied Informatics of the Federal Rural University of Pernambuco, as a partial requirement to obtain the title of Master in Applied Informatics.

ADVISOR: Prof. Dr. Ermeson Andrade

RECIFE – PE

2025

Dados Internacionais de Catalogação na Publicação
Sistema Integrado de Bibliotecas da UFRPE
Bibliotecário(a): Auxiliadora Cunha – CRB-4 1134

F383i Ferreira, Saulo Lucas Gomes.
Impactos de desempenho dos níveis de consistência em bancos de dados NoSQL considerando diferentes ambientes / Saulo Lucas Gomes Ferreira. – Recife, 2025.
51 f.

Orientador(a): Ermeson Andrade.

Dissertação (Mestrado) – Universidade Federal Rural de Pernambuco, Programa de Pós-Graduação em Informática Aplicada, Recife, BR-PE, 2025.

Inclui referências, apêndice(s) e anexo(s).

1. Banco de dados. 2. Banco de dados não relacionais. 3. Banco de dados - Controle de qualidade. 4. Desempenho - Avaliação 5. Computação em nuvem. I. Andrade, Ermeson, orient. II. Título

CDD 004

Acknowledgements

I would like to thank my wife, Maryane, for all the support throughout this journey, for lifting me up during my lows and helping me focus on my dreams. This way would be even harder without her by my side. I also express my deep gratitude to my parents, Katia and Sergio, who always motivated me in my academic career, even when they do not fully understand what some of those things mean. We do not make science alone, it is made possible by the people who support and believe in the importance of our work, bringing us to where we are today.

In addition, I extend my thanks to the entire UFRPE, the centenary institution that makes me proud to stand as a model of public education in this country. We are constantly striving to reinforce its social importance, preserve its history, make it more accessible, and expand opportunities for even more people who wish to contribute to science.

I also would like to thank Prof. Dr. Júlio Mendonça, Prof. Dr. Bruno Nogueira and Prof. Willy Tiengo for all their collaboration in my work and projects. In conclusion, I would like to express my sincere gratitude to Prof. Dr. Ermeson Andrade for the invaluable knowledge and support provided throughout this guidance since my graduation. All of this has been absolutely essential to my development.

I have never walked alone. I am grateful to God for this.

Abstract

The use of non-relational databases (DBs) occurs for a variety of reasons, such as storing unstructured information, handling large amounts of data, and managing interdependencies between data that are not related or referenced. To increase availability and reduce latency, database management systems (DBMSs) are often deployed across multiple instances. These instances store replicas and must maintain data consistency. A strongly consistent DBMS ensures that data and replicas remain identical by synchronizing changes with every operation, guaranteeing all instances commit updates successfully. In contrast, a weakly consistent DBMS prioritizes performance and availability, allowing temporary divergence between replicas and ensuring eventual synchronization. As different applications have unique requirements, understanding the trade-offs of consistency levels is crucial. The goal of this work is to evaluate the impact of consistency configurations in three DBMSs (Cassandra, MongoDB, and Redis) on system performance, using different workloads executing queries on the databases and extracting relevant metrics. The experiments showed that Cassandra experiences up to a 200% increase in the average response time of queries when configured for strong consistency in a controlled local environment. At the same time, Redis exhibits a reduction of up to 92% in throughput in the experiment conducted in the cloud. The findings obtained through the results provide insights into trade-offs between performance and consistency in different deployment configurations, whether locally in an internal network or in the cloud, running across multiple regions.

Keywords: Database. NoSQL. Data Consistency. Performance Evaluation. Cloud

List of symbols

DB	*Database*
DBMS	*Database Management System*
GCP	*Google Cloud Platform*
NoSQL	*Not-only SQL*
SQL	*Structured Query Language*
VM	*Virtual machine*
YCSB	*Yahoo! Cloud Serving Benchmark*

Contents

1 Introduction

The distributed system concept consists in utilizing multiple computer instances (nodes) working in a common workflow, sharing responsibilities and parallelizing processes (LANN, 1977). In terms of processing capacity, it adds a modularity layer to systems architecture making possible to increase and decrease the overall system performance by changing the set of instances which compose the group. In addition to this, a set of distributed nodes can also be used for replicate data in a database cluster, to ensure data safety for failure prevention, or even increase availability, as the data can be accessed even if some system nodes come to fail.

In a cloud scenario, those nodes can be distributed globally, to preserve data even in a natural disaster situation or reduce response time in requests for clients located far from each other (BHAGWAN et al., 2003). In a replicated database system, the replica nodes should always have the same data, even in the moments of writing operations, the data modification applied in a node should be propagated through the replicas to ensure the system's data consistency (ÖZSU; VALDURIEZ, 1996). The called strong consistency database systems respond the same data independent of which node handles the request, since the system validates if all replica nodes committed successfully the operation. This workflow can increase the response time to the system, as the coordinator node awaits every other node before respond the request (ABADI, 2012).

However, in a cloud distributed system, the communication is affected by network instability and it is intensified by the latency between nodes, specially when the cluster is deployed in multiple availability zones. In this case, ensuring all the nodes persisted every operations can affect severely the system performance. For applications requiring quick responses in real-time update scenarios, it is more common to use an eventual consistency configuration. In this setup, the database marks an operation as successful after applying the change to the coordinator node, while secondary instances are updated asynchronously in the background (BURCKHARDT, 2014). This configuration does not ensure that the data returned is the most up-to-date data version, but it indicates that eventually will be. Therefore, systems must take into account trade-offs between consistency and performance to evaluate which configuration fits better to the application specific requirements.

Network-based issues may be mitigated when the system adopts a distributed

database architecture with nodes deployed within a local cluster. By sharing a common local area network with nearby instances, network instability and latency can be reduced (CHARYYEV et al., 2020). This configuration enhances communication between database management system (DBMS) nodes, improving synchronization processes. Such an approach is particularly advantageous in scenarios requiring high-frequency data updates or low-latency responses, as it minimizes the delays typically associated with wide-area networks, ensuring faster and more reliable system performance.

To explore the impacts of consistency configurations, some works have tested NoSQL databases under various conditions in different environments. WANG et al. evaluated how the latency changes when the consistency e replication levels is adjusted for HBase and Cassandra. In their benchmark test using the Yahoo! Cloud Serving Benchmark (YCSB), they observed that Cassandra's write latency increased significantly when the strongest consistency level was used, and that increasing the number of replicas negatively impacted read performance in both databases. In addition, FERREIRA et al. adopted Locust, a Python-based load testing tool, to simulate real-world web traffic on requests executed to locally-deployed Cassandra nodes, varying some parameters, as number of concurrent users and consistency level. The results showed that the strongest consistency level can require over 100% more time to process requests than the weakest consistency level. However, these studies often have limitations, such as the lack of comparisons between DBMSs, limited consideration of multiple workloads, and insufficient metric analysis.

This master's thesis presents two works that conduct an in-depth analysis of how multiple DBMSs respond to different consistency scenarios when exposed to varying workload conditions. These studies provide valuable insights into the trade-offs between performance and consistency, demonstrating how these factors fluctuate based on the deployment environment. Furthermore, the findings help establish criteria for determining which DBMS is best suited for specific use cases, ensuring optimal system performance under different operational conditions.

1.1 Objectives

This work aims to analyze how consistency configurations influence the performance of Cassandra, MongoDB, and Redis in terms of response time and throughput by evaluating

their behavior under varying workloads in both local and cloud-based environments. The specific objectives are:

1. Analyze the impact of different consistency levels on the response time and throughput of Cassandra, MongoDB, and Redis when subjected to various workloads.
2. Evaluate the influence of deployment environments by comparing performance results in local, single-region cloud, and multi-region cloud setups.
3. Compare the behavior of multiple DBMSs under similar conditions to identify the strengths and weaknesses of each system.

1.2 Structure of the Document

Section 2 introduces the articles that support this thesis, highlighting their main contributions and results. Section 3 concludes the work by presenting the contributions, limitations, and potential directions for future research.

2 Articles Supporting this Thesis

This chapter presents the works developed and published to support the main objective of this thesis. All the articles discussed the primary impacts of consistency levels on DBMS performance, exploring various deployment scenarios, including local and cloud-based testbeds.

2.1 Articles Compiled in this Thesis

This section summarizes the main objectives and contributions of each work compiled in this thesis. Each subsection is titled after the journal where the corresponding work was published.

2.1.1 Journal of Cloud Computing

Ferreira et al. «Impacts of data consistency levels in cloud-based NoSQL for data-intensive applications». In *Journal of Cloud Computing*. 2024, p. 158.

This article evaluated the impacts of consistency level configurations on the performance of DBMSs. For this purpose, we selected three of the most popular NoSQL databases, according to the DBEngineRanking (DB-ENGINES, 2024). The databases were deployed on a cloud-based platform distributed across three nodes, with each node replicating data to the others. Since communication between nodes is a key part of the replica synchronization process, we focused on how the distance between nodes affects the metrics obtained from the experiments. This was made by executing all the test scenarios with two deployment types: one in a single availability region, where all nodes are located in different zones within the same region, and one in multiple availability regions, where each node is deployed in a separate region. The environment was set up by instantiating multiple virtual machines (VMs) using the Compute Engine service of Google Cloud Platform (GCP) and the load tests were executed by Locust (HEYMAN et al., 2012), a Python-based framework that simulates real-world user traffic by launching multiple virtual users to make requests to the database.

Key findings demonstrate that while consistency switching in single-region

deployments minimally impacts Cassandra write and MongoDB read performance, it drastically degrades all database operations in multi-region deployments, where inter-node latency becomes a dominant factor. In terms of response time, strong consistency degrades performance by 110% compared to weak consistency in single-region deployments, whereas this degradation exceeds 3,000% in multi-region scenarios.

2.1.2 Software: Practice and Experience

Ferreira et al. «Experimental Performance Analysis of Data Consistency Levels in NoSQL Databases». In *Software: Practice and Experience*. 2025.

This work evaluated the performance of NoSQL DBs with different consistency levels. However, unlike the previous work, this one instantiated the testbed in a local environment simulating a cluster. The experimental setup consisted on two physical machines running a client and the database instances. While the first machine uses Locust to execute the test scenarios with multiple workloads, the second machine runs three database instances (one primary and two secondary nodes) executing on Docker containers with resources limits and connected to each other for data replication. We used Linux Traffic Control (ALMESBERGER et al., 1999) to introduce latency in the communication between the nodes, addressing the limitation of running all nodes on a single machine.

Key findings revealed that Cassandra is the most affected DBMS, with the most response time increasing and the most throughput drop, for both read and write operations. The least affected DBMSs are Redis in writes and MongoDB in reads. However, Redis reads were not evaluated on the experiments because it does not allow consistency configuration for this operation.

2.2 Statement of Authorship

The author of this dissertation is the principal author of the works presented in this chapter. A detailed description of the author's contribution to each work can be found in the corresponding appendices. None of the works in this compendium have been included in other dissertation compendiums, nor will they be included in such compendiums in the future.

3 Final Considerations

This thesis evaluates how various NoSQL databases are affected by multiple consistency configurations in terms of response time and throughput. The analysis offers valuable insights for planning the deployment of a DBMS. Software architects and engineers can align application requirements with the trade-offs identified, helping to decide when ensuring consistency is worthwhile, even if it comes at the expense of some performance. To achieve the results discussed in the presented papers, different testbeds were deployed in both local and cloud-based clusters, with each database configured across multiple nodes.

Local experiments revealed that, under strong consistency, DBMS response times can increase by up to 200% compared to the weakest consistency configuration. Redis was the least affected, with write operations increasing by approximately 10%. Cassandra, the most affected, saw response time increases of up to 200% for both read and write operations. In the cloud-based experiments, systems deployed in single and multiple regions were used to evaluate the impact of latency between nodes located far apart on the performance of a strongly consistent system. In single-region scenarios, average response times increased by 110% for reads and 78% for writes. In multi-region scenarios, these increases rose to approximately 170% for reads and 3,400% for writes.

The findings suggest that Redis is the most suitable database for performance-driven scenarios where eventual consistency is acceptable, as it consistently exhibited the lowest response time across all cases, despite being more sensitive to consistency constraints. This makes Redis a better fit for applications prioritizing availability and quick response times, such as social networks, caching, or messaging apps. These applications can tolerate slightly outdated information, as it will be quickly updated. The results also indicate that MongoDB is better suited for systems requiring strong consistency across all nodes, even if some nodes fail. For instance, a banking application would benefit from this feature, as it requires real-time synchronization to validate operations and ensure their accuracy. Lastly, Cassandra, with its masterless architecture, is ideal for applications that need consistent long-range distributed nodes, particularly for read-heavy operations with low latency. This makes it well-suited for use cases like international financial services and logging history applications.

3.1 Contributions

This work makes contributions to the field of NoSQL DBMS evaluation, particularly regarding the impacts of consistency configurations on performance. The main contributions include:

1. **Comprehensive Evaluation of NoSQL DBMSs:** The study compares the impact of different consistency configurations on Cassandra, MongoDB, and Redis DBMSs, covering both local and cloud-based scenarios. This analysis provides a deeper understanding of how each system handles the trade-off between consistency and performance.
2. **Testing in Local and Cloud Environments:** Experiments conducted in local clusters and the cloud allow for a robust evaluation under various latency and geographic distribution conditions, helping to understand how the systems behave in real-world scenarios, both in terms of performance and scalability.
3. **Insights for DBMS Deployment:** The findings offer valuable insights for software architects and engineers when planning DBMS deployment. The study emphasizes situations where sacrificing some performance in favor of consistency may be advantageous, providing guidance on selecting the best configurations based on the specific requirements of each application.
4. **Practical Recommendations:** Based on the results, the study provides practical recommendations for choosing the most suitable DBMS for different types of applications, such as social networks, banking services, and international financial services systems

3.2 Limitations

This section discusses limitations of the articles and experimented presented in this work.

1. **Availability**: The CAP theorem (BREWER, 2000) discusses the fundamental trade-off between availability and consistency in a partitioned system, arguing that a system cannot achieve both simultaneously. To narrow the focus of this master's thesis, the experiments presented focus on consistency changes, particularly performance, rather than exploring availability.

2. **Limited resources**: Due to limited resources, we were unable to explore scenarios involving multiple data centers and clusters, even though Cassandra and Redis offer specific configurations for such deployments. The lack of local compute instances and the limitations of the GCP free-tier made this exploration unfeasible.
3. **Unique target host**: In the experiments, we designated a single node to handle the requests, acting as the coordinator for every operation executed on the database. Although Cassandra's masterless architecture allows any node to act as the coordinator, we chose to designate just one node to standardize the tests. This configuration was compatible with MongoDB and Redis, which could also adopt this setup within the testbed we had available.

3.3 Future Works

The analysis presented here highlights numerous possibilities for further exploration. Since this work focuses on consistency levels, there is room to study in detail how the system's overall availability is affected by different types of configurations. This analysis could further explore the complete CAP theorem and measure the expected consequences across different NoSQL databases.

Besides the impacts on latency and throughput, the consistency approach can also be analyzed in terms of how long the data remains outdated before the DBMS synchronizes it in an eventual consistency scenario. The tests could measure the probability of a request returning outdated data when there is a series of update operations on the database. This experiment could serve as an addition to the results discussed in this work and provide further arguments for the discussion of consistency configuration trade-offs.

Lastly, the experiments would provide even more meaningful results if conducted on machines with greater CPU and memory resources, more closely reflecting real-world database instances. Beyond hardware resources, the experiment could benefit from additional compute instances to create multiple data centers and clusters with internal replication. This would allow for the exploration of different database configurations suited to this specific system architecture and a deeper analysis of how these configurations impact system performance under each consistency setting discussed in this work.

Bibliography

ABADI, D. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. **Computer**, IEEE, v. 45, n. 2, p. 37–42, 2012.

ALMESBERGER, W. et al. **Linux network traffic control—implementation overview**. 1999.

BHAGWAN, R.; SAVAGE, S.; VOELKER, G. M. Understanding availability. In: SPRINGER. **International Workshop on Peer-to-Peer Systems**. [S.l.], 2003. p. 256–267.

BREWER, E. A. Towards robust distributed systems. In: PORTLAND, OR. **PODC**. [S.l.], 2000. v. 7, n. 10.1145, p. 343477–343502.

BURCKHARDT, S. Principles of eventual consistency. 2014.

CHARYYEV, B.; ARSLAN, E.; GUNES, M. H. Latency comparison of cloud datacenters and edge servers. In: IEEE. **GLOBECOM 2020-2020 IEEE Global Communications Conference**. [S.l.], 2020. p. 1–6.

DB-ENGINES. **DB-Engines Ranking**. 2024. <https://db-engines.com/en/ranking>. [Online; accessed 20-jan-2024].

FERREIRA, S.; ANDRADE, E.; MENDONÇA, J. Uma abordagem experimental para avaliar os níveis de consistência do banco de dados nosql cassandra. In: SBC. **Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho**. [S.l.], 2021. p. 156–167.

HEYMAN, J.; BYSTRöM, C.; HAMRéN, J.; HEYMAN, H. **Locust.io**. 2012. Disponível em: <https://locust.io/>.

LANN, G. L. Distributed systems-towards a formal approach. In: TORONTO. **IFIP congress**. [S.l.], 1977. v. 7, p. 155–160.

ÖZSU, M. T.; VALDURIEZ, P. Distributed and parallel database systems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 1, p. 125–128, 1996.

WANG, H.; LI, J.; ZHANG, H.; ZHOU, Y. Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra. In: SPRINGER. **Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware**. [S.l.], 2014. p. 71–82.

APÊNDICES

APPENDIX A – Impacts of data consistency levels in cloud-based NoSQL for data-intensive applications

Ferreira et al. «Impacts of data consistency levels in cloud-based NoSQL for data-intensive applications». In *Journal of Cloud Computing*. 2024, p. 158.

Author's contribution: The author was responsible for developing methodology, experiment setup and execution, data processing, resource generation (including tables, charts, and images), and writing the original draft.

Ferreira *et al. Journal of Cloud Computing (2024) 13:158*
https://doi.org/10.1186/s13677-024-00716-7

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Impacts of data consistency levels in cloud-based NoSQL for data-intensive applications

Saulo Ferreira[1*], Júlio Mendonça[2†], Bruno Nogueira[3], Willy Tiengo[3] and Ermeson Andrade[1]

Abstract

When using database management systems (DBMSs), it is common to distribute instance replicas across multiple locations for disaster recovery and scaling purposes. To efficiently geo-replicate data, it is crucial to ensure the data and its replicas remain consistent with the same and the most up-to-date data. However, DBMSs' inner characteristics and external factors, such as the replication strategy and network latency, can affect system performance when dealing with data replication, especially when the replicas are deployed far apart from the others. Thus, it is essential to comprehend how achieving high data consistency levels in geo-replicated systems can impact systems performance. This work analyzes various data consistency settings for the widely used NoSQL DBMSs, namely MongoDB, Redis, and Cassandra. The analysis is based on real-world experiments in which DBMS nodes are deployed on cloud platforms in different locations, considering single and multiple region deployments. Based on the results of the experiments, we provide a comprehensive analysis regarding the system throughput and response time when executing reading and writing operations, pointing out scenarios where each DBMS could be better employed. Some of our findings include, for instance, that opting for strong data consistency significantly impacts Cassandra's reading operations in the single-region deployment, while MongoDB writing operations are most affected in a multi-region scenario. Additionally, all of these DBMSs exhibit statistically significant variations across all scenarios in the multi-region setup when the data consistency is switched from weak to stronger level.

Keywords Cloud, Data consistency, Databases, NoSQL, Performance

Introduction

The idea of designing distributed systems predates the advent of cloud-based services. Even before the rise of platforms like Amazon Web Services (AWS) and Google Cloud Platform (GCP), the model of horizontal scalability existed, aiming to parallelize processes across multiple processing cores [1]. Horizontal scalability contrasts with vertical scalability, which involves increasing resources within the same computational system [2]. Nowadays, cloud computing has facilitated both horizontal and vertical scalability of applications. Beyond that, cloud computing made distributed systems more accessible and widespread. It boosted strategies such as geo-replication, where computational instances (a.k.a. nodes) can be deployed in different physical locations. Geo-replicated systems have been widely adopted for disaster prevention and recovery [3], fault-tolerance [4], and latency reduction in communication between clients and cloud servers [5].

†Júlio Mendonça, Bruno Nogueira, Willy Tiengo and Ermeson Andrade contributed equally to this work.

*Correspondence:
Saulo Ferreira
saulo.gomesferreira@ufrpe.br
[1] Universidade Federal Rural de Pernambuco, Recife, Pernambuco, Brazil
[2] Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg, Luxembourg
[3] Universidade Federal de Alagoas, Maceió, Alagoas, Brazil

Modern Database management systems (DBMSs) adopt the concept of replication for distributing data among different nodes that can be physically located in the same region or geo-replicated [6]. The distributed architecture prevents the entire system from becoming unavailable and inaccessible in the event of a failure in a subset of nodes. Additionally, it enables end-users (or client applications) to access data more quickly from a node closer to them [7]. However, (geo-)replicating data can lead to data inconsistency. When a specific piece of data is updated on an arbitrary node *A*, the same piece of data might be requested on another arbitrary node *B*, which may not have been updated yet. Some DBMSs address this issue by employing a strong consistency scheme, where a node can only confirm the update of a particular piece of data to the client application when all nodes in the cluster confirm they received and updated this same piece of data [8]. This approach may compromise performance (e.g., the response time in the mentioned example) for the sake of data consistency [9, 10], especially when the replica nodes are located in a different place when the distance delay affects the communication more significantly.

Systems prioritizing performance often implement eventual data consistency (or weak data consistency), where data updated on one node A may not immediately reflect on a second node B but will eventually synchronize. An asynchronous data update method is employed to achieve this, returning success for a request while updating other replicas in the background [11]. The aim is to maintain data and its replicas up-to-date at some point, though not necessarily simultaneously. Consequently, these systems have a distinct trade-off between data consistency and performance, highlighting the importance of evaluating the compromise between the two aspects.

Previous works have explored various techniques to evaluate data consistency levels for databases (DBs) in terms of performance. For instance, Gorbenko et al. [9] used a benchmark evaluation to compare different consistency levels of the Cassandra database, aiming to understand the performance costs associated with each assessed level. On the other hand, Diogo et al. [12] analyze how different non-relational DBMSs - namely Cassandra, MongoDB, OrientDB, and Redis - are implemented to address data consistency issues. It offers an overview of replication and synchronicity configurations in a distributed architecture with multiple nodes. However, these studies often have limitations, such as the need for comparison between DB systems, challenges in applying workloads, and limited metrics analysis.

In this manner, this study conducts a performance analysis of multiple NoSQL DBs deployed in a cloud-based environment in terms of response time and throughput. Our analysis focuses on scenarios where nodes are deployed within single- or multiple-cloud regions. The main objective is to identify how the response time and throughput of each NoSQL DBMS are affected by different data consistency levels, workload conditions, and deployment locations. This allows an evaluation of how the geographical distance between nodes can affect the synchronization of data persisted in NoSQL DBMSs. We evaluated three widely adopted NoSQL DBMSs, namely Cassandra, MongoDB, and Redis, deployed in a cloud environment with the same hardware resources to handle diverse loads of concurrent requests from an external system. By subjecting these DBMSs to identical load scenarios across different consistency levels, we assess the performance implications of each data consistency level on the system's operation. Our findings show that Cassandra and Redis persist in writing operations faster and are less impacted by data consistency level changes, while MongoDB has better response time for reading operations and is more affected by data consistency level changes in writing operations.

Thus, the main contributions of this work are:

- We set geo-distributed NoSQL DBMSs in a real-world cloud environment to perform experiments considering different scenarios with concurrent users, data consistency levels, and DB operation;
- We analyze how performance metrics (i.e., response time and throughput) of different DBMSs are affected by different configurations for ensuring data consistency;
- We identify the best and worst operations in terms of response time and throughput for each DB under various data consistency levels and workload conditions;
- We analyze the statistically significant differences regarding the throughput and response time of writing and reading operations of different DBMSs;
- We compare how the deployment of distributed DBMSs in single- and multi-region on clouds can impact the throughput and response time.

The remainder of the paper is organized into six main sections. "Background" section presents the key concepts essential for understanding the work. "Related works" section presents the related work. "A cloud-based NoSQL environment" section describes the experimental setup used to conduct the experiments. "Experiments results and discussion" section discusses the obtained results. Finally, "Conclusions and future work" section concludes the paper and briefly introduces future work.

Background

This section introduces the most relevant concepts addressed in this work.

NoSQL databases

Non-relational (NoSQL) DBs diverge from traditional relational DBs in their data storage models. Unlike relational DBs, which organize data into tables with predefined schemas, NoSQL DBs utilize flexible schemas, allowing for dynamic and unstructured data storage [13]. They are particularly relevant for the analysis adopted in this work because they are built to handle large volumes of data with reduced impact on latency and processing time, prioritizing performance in distributed deployment scenarios [14]. These types of DBMS are often deployed in clusters with multiple computational nodes to distribute the data and ensure that replicas can operate independently in the event of failure occurring in any of them. Although there are several NoSQL DBs available, we have selected three of the most popular ones, according to the DBEngine Ranking [15]: Cassandra, MongoDB, and Redis. These DBMSs offer different features and capabilities, making them suitable candidates for investigating various aspects of performance and data consistency in distributed environments.

When a NoSQL DB is configured to replicate data across different nodes, it must address the issue of update synchronization. A NoSQL DB focused on availability, for example, typically prioritizes asynchronous updating of replica data to avoid impacting system performance, opting instead for eventual consistency. On the other hand, a NoSQL DB focused on consistency aims to keep the data as up-to-date as possible, updating it during requests and confirming an operation only when all replicas are updated. This duality is described in Dr. Eric Brewer's CAP theorem [16], which explains that fault-tolerant distributed systems must choose between the two principles of availability and consistency in the face of partition failures [17].

Cassandra

Cassandra [18] is a performance-focused NoSQL DBMS based on tables, like traditional relational DBs. Designed for distributed operations across hundreds of nodes, Cassandra adopts a masterless architecture, eliminating the need for a centralizing main node. Instead, each node can function as a request coordinator. To ensure fault tolerance, Cassandra natively supports deployment across multiple data centers, enabling data replication over a customizable number of nodes. Users have the flexibility to define the number of replicas in a cluster or data center, as well as whether replicas will be updated synchronously at the time of request or asynchronously in parallel subprocesses. This adaptability allows for configuring Cassandra, which prioritizes availability, to achieve a higher level of consistency in the system, even at the expense of performance.

To configure the system's consistency level, Cassandra offers various configurable options, including 'ONE', 'QUORUM', 'ALL', and 'ANY'. However, to standardize across all the DBMSs we chose, we summarize the main levels in terms of weak and strong consistency as follows:

- **Weak**: Utilizing the *ONE* consistency level for reads, Cassandra allows the possibility of retrieving the data replica stored on the node closest to the request coordinator, without guaranteeing it to be the most current in the system. Regarding writes, the *ONE* level permits the system to update data and complete the operation awaiting updates from only another replica.
- **Strong**: Employing the *ALL* consistency level for reads, Cassandra mandates validation of the requested data across all replicas to ensure retrieval of the most current version available at the time of the request. For writes, Cassandra requires all replicas to be updated before returning success in the operation.

MongoDB

MongoDB [19] is a document-based NoSQL DBMS that utilizes the BSON format (similar to JSON) for storing data identified by an object ID. Operations on this DB can be performed using a command-line interface, facilitating quick insertion and retrieval of documents organized into collections. While collections in MongoDB serve a purpose similar to tables in a relational SQL DB, they do not adhere to a predefined schema. Consequently, there is no assurance that two documents stored in the same collection have identical formats. As a result, collections serve more as aggregators of documents with similar structures or data models rather than as a set of uniform objects.

In its architecture, MongoDB adopts the primary-secondary model by default, where one node is responsible for handling requests and managing data, while the others act as replicas. This model is structured so that, in the event of a failure in the primary node, the system can promptly elect a new primary from the available secondaries. Despite operating with this scheme of centralized responsibilities, MongoDB also offers a load balancing mechanism. This facilitates the distribution of large volumes of data across different nodes to manage high-demand operations using horizontal scalability.

To maintain consistency among distributed and replicated data, MongoDB allows users to configure weak and strong consistencies using settings for reading and writing concerns. These concerns refer to the level of guarantee MongoDB provides regarding the consistency of data reads and writes across distributed nodes. Below, we present the concerns adopted in this work, referred to as weak and strong consistency.

- **Weak**: By using the "local" reading concern for reads, the user directs MongoDB to confirm the version of the data present in the closest replica, without considering whether more updated versions exist in other replicas. Regarding writes, the writing concern parameter enables the user to specify how many replicas should persist the change, with 1 typically indicating the closest validation only.
- **Strong**: Opting for the "linearizable" reading concern for reads instructs MongoDB to return data reflecting all successful writes in the system. For writes, using the aforementioned writing concern parameter, the user can specify the total number of replicas in the system to indicate that the operation should be persisted only after all replica set members confirm success.

Redis

Redis [20] is a multi-purpose, open-source DBMS used in a variety of applications, including real-time social media analytics, ad targeting, and caching. Due to its straightforward, non-structured architecture, Redis is often characterized more as a data storage structure than a traditional DB. It supports several data types organized in a key-value mapping, facilitating quick retrieval via a command-line interface, SDK, or API. One of Redis's notable features is its ability to operate entirely in memory, making it a preferred choice for real-time responsive applications. This characteristic enhances its capacity to provide instantaneous access storage [21].

While Redis primarily emphasizes availability, it also offers configuration options for consistency levels. It supports the *WAIT* command, which enables setting synchronous replication for write operations. This feature enhances system consistency by specifying the number of replicas that must confirm the operation before the request is considered complete. As Redis does not support consistency levels for reads, in this work, we focus solely on the consistency level adopted for write operations, as presented below:

- **Weak**: Utilizing the "WAIT 1" command in write operations prompts the system to validate success by ensuring the successful persistence of the written data on only one replica.
- **Strong**: Employing "WAIT N" – where "N" represents the number of replica nodes – prompts the system to await confirmation of success from all replica nodes before completing the request.

Related works

NoSQL DBs have seen widespread adoption in the past decade due to the exponential growth of data usage for data-intensive applications, such as Big data and training of artificial intelligence models. Previous research has explored different aspects of these systems. While existent surveys provide a summary of the background and main characteristics of NoSQL DBMSs [22–24], various works have been conducted to evaluate NoSQL DBMSs performance. However, there remains a limited focus on analyzing DBMSs' data consistency levels in terms of performance. Next, we explore studies that specifically target performance or data consistency levels evaluation in NoSQL DBMSs. Then, we highlight how our work fills current research gaps in the existing literature.

Some studies have analyzed and compared the performance of NoSQL DBs. Abu Kausar et al. [25] utilized the YCSB (Yahoo! Cloud Serving Benchmark) to evaluate the performance of MongoDB, Cassandra, and Redis. In this study, Cassandra exhibited the lowest throughput among these DBs, while MongoDB demonstrated superior performance across nearly all executed workloads. Gandini et al. [26] conducted a benchmarking analysis for MongoDB, Cassandra, and HBase on the cloud was presented, with HBase outperforming the others in almost all cases. Abramova et al. [27] presented an evaluation of five DBs on a local virtual machine experimental setup, where OrientDB displayed the poorest overall performance compared to MongoDB, HBase, Cassandra, and Redis, which performed the best.

It is also possible to find works that have explored the evaluation of data consistency levels. Wang et al. [28] compared the DBMSs Cassandra and HBase, considering a different number of replicas and consistency levels. They also conducted comprehensive stress benchmarks to evaluate the performance of each DBMS when handling read and write operation requests under heavy loads. Gomes et al. [29] developed an approach based on Petri nets to evaluate the consistency and availability of a three-node Cassandra cluster. That work revealed that the number of replicas and the response time of the coordinator node are the main reasons for decreasing the DB performance. Haughian et al. [30] analyzed the throughput of Cassandra and MongoDB across various consistency levels, observing changes in the number of operations per second as the number of nodes required

for coordinator operation varied. The study's findings revealed that MongoDB's master-slave configuration is effective in reducing replication impacts, with writing operations being more significantly impacted at higher workloads. However, in a non-replication scenario, the authors showed that Cassandra can scale better than MongoDB.

Gorbenko et al. [9] proposed a benchmark evaluation of different Cassandra's consistency levels deployed on AWS EC2. The main objective of that work was to study how different consistency levels affect Cassandra's performance through high workloads. It revealed that strong consistency can decrease performance by 25% and the biggest impact is on read operations. Additionally, they proposed a set of optimized consistency settings for Cassandra that can maintain strong consistency while maximizing performance. Ferreira et al. [31] investigated the consistency of Cassandra and analyzed the associated latency across different consistency levels. The results revealed a growth superior to 100% in the latency (from the weakest to the strongest consistency level) under different workloads.

Unlike previous studies, this work aims to evaluate a multi-node cluster deployed on a geo-distributed and cloud environment, considering three different DBMSs: Cassandra, MongoDB, and Redis. Table 1 presents a comparison of the main characteristics of related works and this paper, demonstrating the significance of our findings in the context of previous research. Existing studies in this domain often present certain limitations, such as a lack of comparison between different DBMSs, challenges in applying realistic workloads, and a narrow analysis of metrics. Therefore, this work has the potential to assist in deploying NoSQL DBMSs across various industries and applications that need to consider the trade-offs among data consistency levels and performance. By generating detailed data on the performance and scalability of Cassandra, MongoDB, and Redis under different configurations, this study provides valuable insights for organizations seeking to optimize their DB environments.

Table 1 Comparison of related works main characteristics

Work	Consistency evaluation	Database comparison	Workload variation	Performance analysis
Han et al. [22]	No	Yes, 7	No	No
Mohamed et al. [23]	No	No	No	No
Abu Kausar et al. [25]	No	Yes, 3	Yes	Yes
Gandini et al. [26]	No	Yes, 3	Yes	Yes
Abramova et al. [27]	No	Yes, 5	Yes	Yes
Wang et al. [28]	Yes	No	No	No
Gomes et al. [29]	Yes	No	No	Yes
Haughian et al. [30]	Yes	Yes, 2	Yes	No
Gorbenko et al. [9]	Yes	No	Yes	Yes
Ferreira et al. [31]	Yes	No	Yes	Yes
This work	**Yes**	**Yes, 3**	**Yes**	**Yes**

A cloud-based NoSQL environment

This section details the cloud-based NoSQL environment adopted in this work for experimental analysis, including the platforms utilized, virtual machine configurations, workloads, and DB input parameters.

The testbed

We deployed our testbed on the Google Cloud Platform (GCP) to leverage different data center locations. We configured four Virtual Machines (VMs) across the data centers and assigned specific roles for each of them, categorized into two groups: the DB nodes (*primary-db-node-a/b* and *secondary-db-node-a#*, *secondary-db-node-b#*) and the workload generator (*client-node*), as shown in Fig. 1.

All the VMs configured as DB nodes had only one of the three selected DBMSs (i.e., Cassandra, MongoDB, and Redis) activated and running during each respective experiment. The DB nodes were categorized into primary and secondary nodes, with the *primary-db-node* serving as the target for client requests and the others (*secondary-db-node-a#*, and *secondary-db-node-b#*) functioning as replication nodes. It is worthwhile to mention that Cassandra adopts a masterless structure, allowing any node to handle requests, but for a fair comparison with all DBMSs, we configured only the *primary-db-node* to handle requests in our experimental setup.

We had two different deployment settings: (a) single-region and (b) multi-region. In single-region deployment (see Fig. 1a), the VMs are located in the same region but in multiple zones of GCP. This configuration reduces communication delays and also isolates VMs against zone-based failures. On the other hand, despite the GCP documentation guaranteeing cluster independence, the proximity to operational centers does not eliminate the risk of failures caused by physical accidents and disasters. This deployment utilized the Iowa region zones (i.e., us-central1) to deploy all four nodes. For the multi-region deployment (see Fig. 1b), physical distance for the node deployments is also considered to mitigate accidents and natural disasters. However,

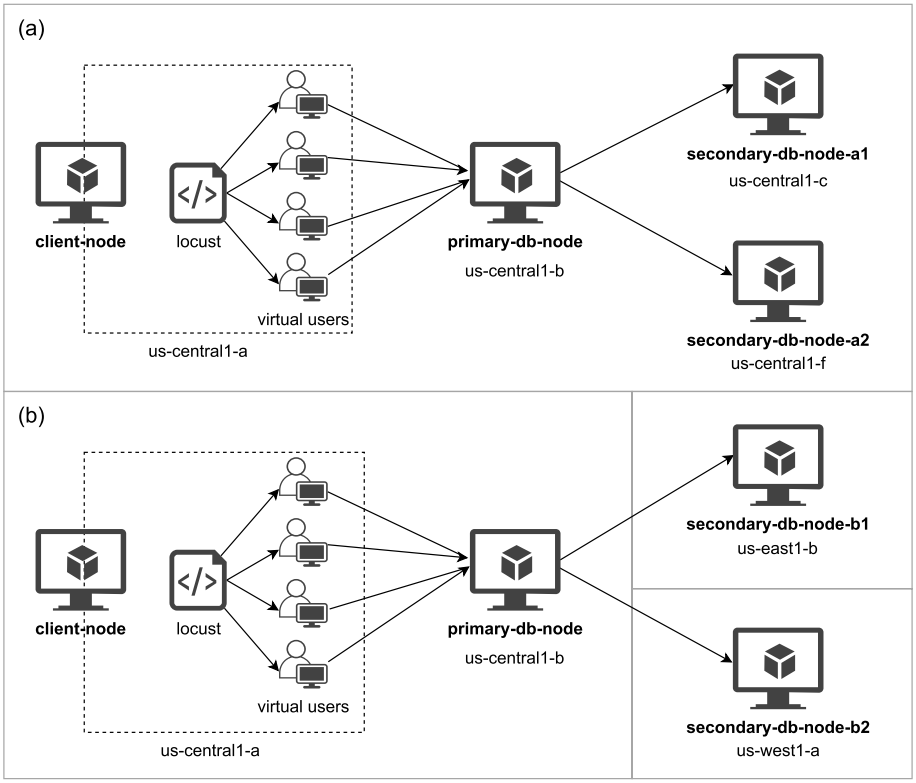

Fig. 1 Adopted testbed deployed on **a** a single-region (us-central1-a/b/c/f) and **a** multi-region (us-central1-a/b, us-east1-b, us-west1-a)

Table 2 VMs default configurations and roles

VM ID	Resource Config.	Zone/Region
client-node	E2-4vCPU/8GB RAM	us-central1-a
primary-db-node-a	E2-2vCPU/4GB RAM	us-central1-b
secondary-db-node-a1	E2-2vCPU/4GB RAM	us-central1-c
secondary-db-node-a2	E2-2vCPU/4GB RAM	us-central1-f
primary-db-node-b	E2-2vCPU/4GB RAM	us-central1-b
secondary-db-node-b1	E2-2vCPU/4GB RAM	us-east1-b
secondary-db-node-b2	E2-2vCPU/4GB RAM	us-west1-a

this setup may introduce communication delays and affect performance. This deployment also used the Iowa region zones to deploy the client and primary DB node but placed replica nodes in the Oregon (i.e., us-east1) and South Carolina (i.e., us-west1) regions. For an easy compression of the experiments configuration, we named the VMs with the suffix "a" when used in the single-region deployment (e.g., *primary-db-node-a*) and with the suffix "b" when used on the multi-region deployment (*primary-db-node-b*). Table 2 summarizes the configurations of each VM adopted on the GCP.

The *client-node* served as the primary source of the workload generation for the experiments in both deployments. It was responsible for executing a multi-threaded script that generates concurrent requests to the DB nodes. To ensure optimal workload generation and accurate test results, the VM assigned as *client-node* was configured with additional resources (4 vCPUs and 8 GB RAM) to mitigate potential limitations that could interfere with the workload generation (see Table 2).

For conducting the experiments, the *client-node* utilized Locust [32] version 2.12.1, an open-source load testing tool for Python. We used Locust due to its capability of simulating swarm-based traffic patterns, reproducing real-world web application loads where multiple users can make concurrent requests [33]. Additionally, it enables the execution of tests with various settings, including the number of concurrent users, time intervals between requests, user spawn rate, and total duration. Table 3 specifies Locust used settings. Given our approach of executing Locust based on a fixed duration, the number of requests per second generated by each user is determined by the system's capacity to handle requests under varying conditions. Each Locust user sends requests sequentially and continuously throughout the specified time frame, so the throughput reflects the

Table 3 Adopted settings in Locust for each experiment conducted

Argument	Value	Description
headless	true	No interface execution.
spawn-rate	10	Users spawn rate.
users	Concurrent users variable	Number of concurrent users.
run-time	300 (seconds)	Duration of test execution.

system's ability to process requests rather than a fixed number of requests per user. For example, if 10 Locust users are active and the system can handle 3,000 requests per second, each user effectively generates an average of 300 requests per second. This adaptive behavior allows us to measure performance trends as the system becomes a bottleneck under higher loads.

Database configurations

We have selected the *weakest* and *strongest* data consistency levels available to evaluate each DBMS. Data consistency level configurations are different for each DBMS (as described in "Background" section). Therefore, we perform experiments using the weakest and strongest for a fair comparison among all DBMSs analyzed. Next, we detail the configuration of each DBMS utilized.

For Cassandra, we deployed the official Docker image version 5.0[1] and configured it across the three DB nodes, employing the default "SimpleStrategy" replication configuration with a replication factor of 3 to ensure data replication across all nodes. We utilized the consistency levels of *ONE* and *ALL* for weak and strong consistency, respectively. We created a table with a single integer unique identifier and ten string columns to read and write data on the Cassandra nodes.

We configured a three-node cluster for MongoDB using the official Docker image version 8.0[2]. Each node serves as both a primary and secondary node, replicating data bidirectionally with the other two nodes. We set the weakest consistency level for reading operations (`read concern`) as *local* and the strongest as *linearizable*. In writing operations, the weakest data consistency configuration (`write concern`) was set to 1, indicating that only one node is necessary to commit the operation. For the strongest data consistency level, it was set to 3, indicating that confirmation from all three nodes is required to commit the operation. Data for reading and writing operations was formatted as a JSON-like document with an integer identifier and ten key-value string pairs.

[1] Cassandra Docker image link

[2] MongoDB Docker image link

Table 4 Parameter values to generate evaluation scenarios

# concurrent users	Consistency level	Operation	Database	Deployment
1-10	weak	read	Cassandra	Single-region
	strong	write	MongoDB	Multi-region
			Redis	

We utilized the official Docker image for Redis, version 7.2[3]. Although this DBMS does not offer full support for data consistency level configurations, it provides synchronous replication as a means to achieve consistency. To enable synchronous replication, the WAIT command is used to specify the number of nodes that should confirm an operation. However, this feature is only available for writing operations. Consequently, we evaluated Redis solely based on writing operations compared to the other DBMSs. We set the `WAIT` command to 1 for the weakest consistency level, and for the strongest, we set it to 3. The data for writing operations consisted of a unique string key and a hash data type to store ten field-value pairs, each containing string values.

For all DBs, the data used for writing operations was the same, following their respective input formats: a randomly generated unique identifier and a set of 10 predefined key-value or column-value data pairs as strings of size. Then, we used the generated data in a `INSERT` statement following the format of each DBMS. Each `INSERT` statement comprised 8,192 bytes. The reading operations consisted of a `SELECT` statement with no filters and a limit of 1000 rows or documents.

Input parameters and evaluated metrics

Our experiments analyzed different scenarios arranged according to five relevant parameters: number of concurrent users, data consistency level, operation, database, and deployment. The number of concurrent users reflects the number of concurrent threads generating continuous requests to the DB with no time interval between them. The data consistency level indicates whether a weak or strong level is employed. The operation indicates if the scenario considers reading or writing operations. The database indicates the database used in the scenario and the deployment if the scenario was in a single- or multi-region deployment setting. Table 4 summarizes the values adopted for each of these parameters to generate evaluation scenarios.

All parameters were utilized to generate a full-factorial collection of scenarios, encompassing all possible

[3] Redis Docker image link.

combinations among them [34]. Each combination of these parameters resulted in an individual scenario, evaluated separately. The total number of combinations was 200, comprising 80 evaluation scenarios for MongoDB, 80 for Cassandra (with variations across 10 concurrency levels, 2 consistency levels, 2 operations, and 2 deployment settings), and 40 for Redis (with variations across 10 concurrency levels, 2 consistency levels, 1 operation, and 2 deployment types). Next, we generated a file containing the scenarios to be evaluated and inputted into the Locust tool. The tool sequentially executed scenarios over a period of 300 seconds, with a one-minute interval between each scenario execution. An example of the generated input file is depicted in Listing 1.

Listing 1 Example of a input file for executing

```
id,users,duration,consistency,operation,database,deployment
1,1,300,weak,reading,cassandra,single
2,2,300,weak,reading,cassandra,single
...,...,...,...,...,...
199,9,300,strong,write,redis,multi
200,10,300,strong,write,redis,multi
```

the experiments with the Locust toolAt the end of the experiments, the Locust tool generated an output file containing metrics for each scenario executed. We collected and analyzed important metrics such as response time, throughput (requests per second), and error rate to understand how performance was affected by data consistency level and workload changes. Additionally, we utilized the GCP Cloud Monitoring agent to collect VM resource usage, including CPU and RAM memory, providing supplementary insights into system performance.

Experiments results and discussion

This section aims to present the results obtained from the experiments. Initially, we present the results related to throughput and response time under different consistency levels and workloads. After that, we examine resource utilization (CPU, memory) on the nodes to identify potential bottlenecks. Next, statistical analysis is conducted to confirm any differences in the results of the adopted metrics. Finally, we present a comparative analysis of our findings.

Throughput and response time analysis

Our analysis examines the system's throughput and response time when implementing the weakest and strongest data consistency levels. We evaluate multi-scenarios for each DBMS and operation to identify how the system is impacted, especially when different data consistency levels are used.

Single-region deployment

This subsection discusses the results of the experiment conducted in the single-region deployment. We first analyze the results for throughput and then, response time. Figure 2 illustrates the throughput of all DBMSs by operation and number of concurrent users. The x-axis of each plot represents the number of concurrent users adopted, while the y-axis illustrates the throughput achieved by the DBMS. The plots also display two distinct lines: the gray line represents the weak data consistency level results, while the black line indicates the strong data consistency level results.

Throughput in writing operations. In general, Redis presents the best writing performance in terms of throughput. In the best-case scenario, it manages to write nearly 4000 requests per second using both weak and strong data consistency levels. On the other hand, MongoDB presented the worst performance, processing nearly 1600 requests per second when adopting the weak data consistency level and around 900 when using the strong data consistency level. Besides, MongoDB exhibits the largest difference between scenarios with weak and strong consistency in writing operations. Specifically, we observe a 64% decrease in the throughput value (from around 694 to 245 requests per second) in scenario with 2 concurrent users. Comparatively, in its worst scenarios, Cassandra shows a 36% decrease (from around 1500 to around 970) with 2 concurrent users, and Redis demonstrates a 25% decrease (from 1389 to 1083) with 2 concurrent users.

Throughput in reading operations. Cassandra's performance is significantly lower compared to writing operations. It experiences around 65% of decrease (from 109 to 37) in the number of requests per second processed by the system in its worst-case scenario with 6 concurrent users. Meanwhile, MongoDB exhibits a 43% decrease (from 160 to 90) in its worst-case scenario with 2 users. The divergence of Cassandra's performance can be attributed to its in-memory storage, which requires runtime write validation across replicas, causing slowdowns in reads under strong consistency. In addition, MongoDB's throughput decreases during the shift from weak to strong consistency, but this decline has minimal

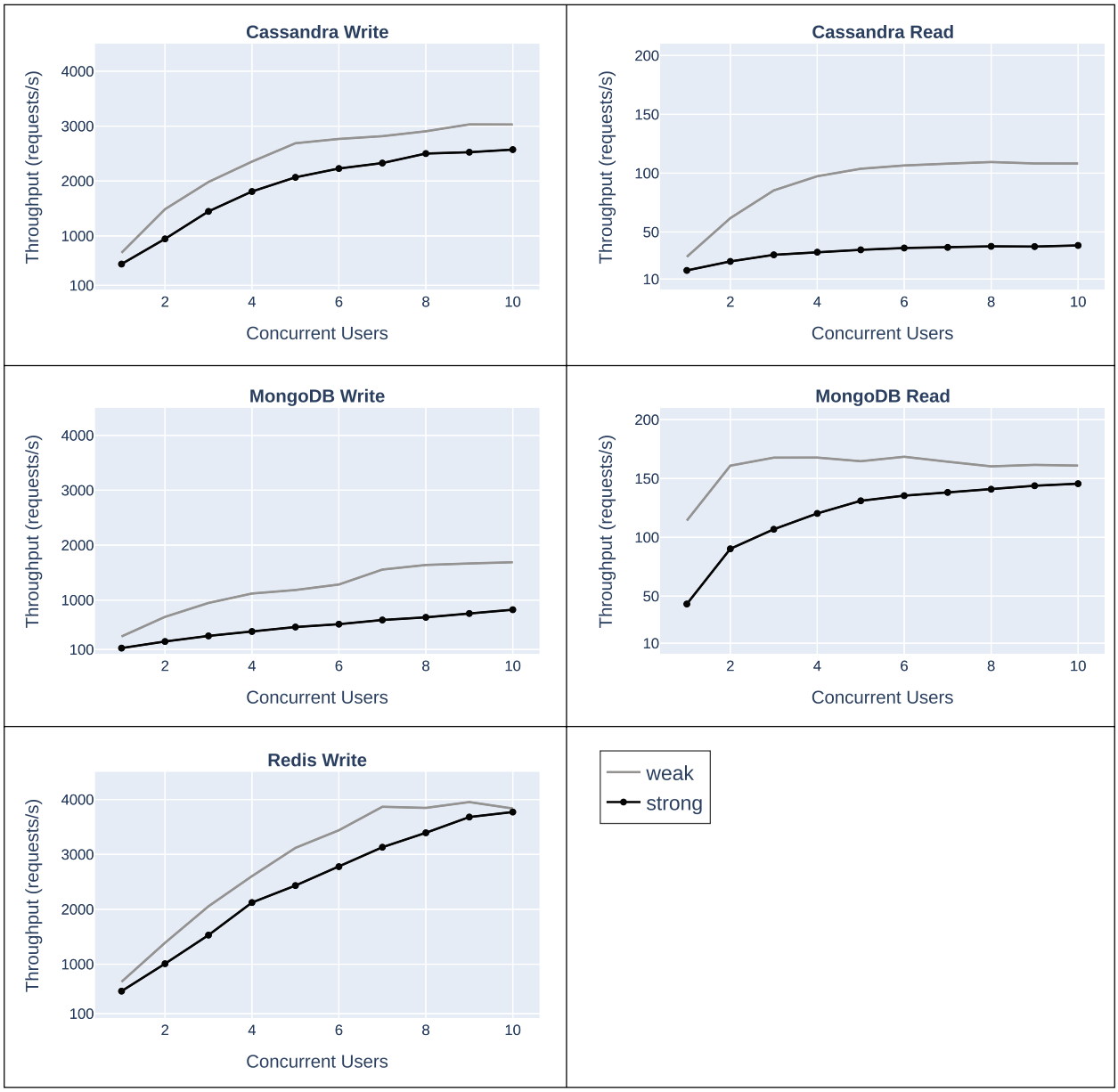

Fig. 2 Throughput results (in requests/s) for writing and reading operations of each NoSQL DBMS analyzed in single-region deployment. A *higher* value in the y-axis means *better* result. The plot scales differ on the y-axis for readability

impact on increasing loads. This behavior suggests that MongoDB may utilize better parallelism compared to the other DBs when operating under high loads.

In the subsequent analysis, we present the results regarding the response time. Figure 3 illustrates the results, with the x-axis representing the number of concurrent users and the y-axis indicating the average response time in milliseconds.

Response time in writing operations. Redis presents the best result, taking a maximum of around 2 milliseconds to write a request. On the other hand, MongoDB can execute writing operations as quickly as the other DBMSs, taking a maximum of 5 milliseconds when adopting the weak data consistency level. However, its performance drops significantly when adopting the strong data consistency level, causing MongoDB to reach an average of 11 milliseconds in the worst-case scenario (with 10 concurrent users). Cassandra and Redis present equivalent increases in the average response time (comparing the weak and strong data consistency levels)

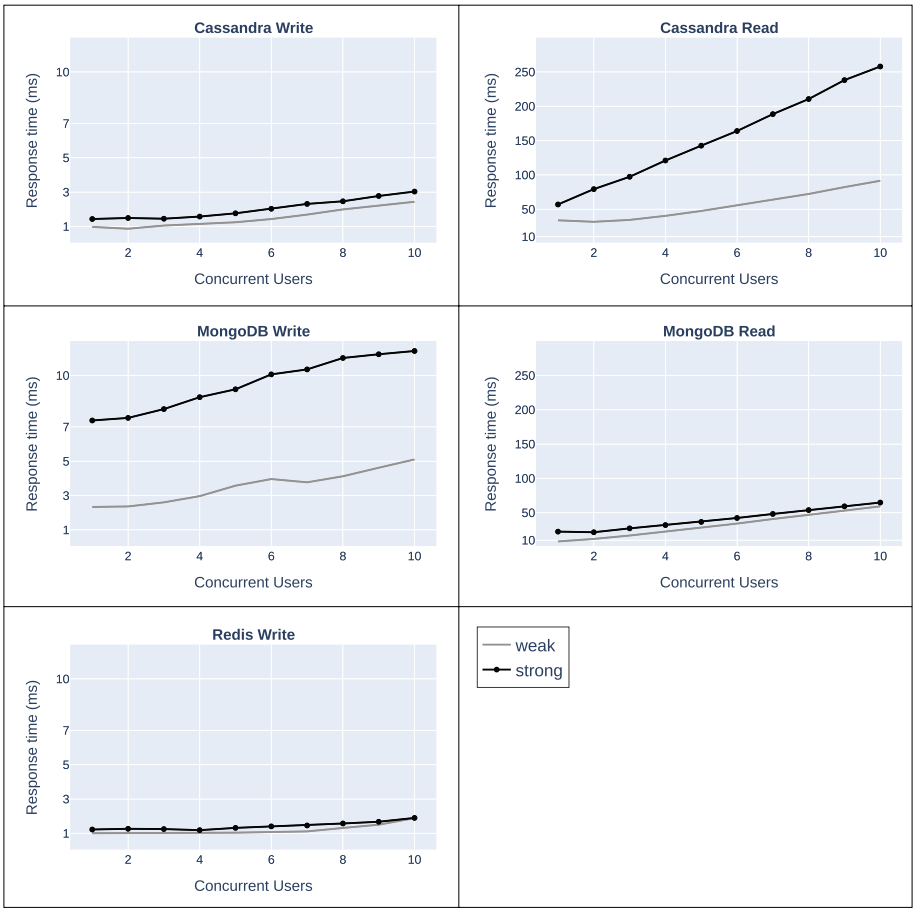

Fig. 3 Response time results for writing and reading operations of each NoSQL DBMS analyzed in single-region deployment. A *smaller* value in the y-axis means *better* result. Scales differ on the y-axis for readability

across the workload scenarios, with Cassandra increasing around 50% and Redis around 25%. MongoDB exhibits the most significant difference between weak and strong consistency results, 218%. Different from the others, Redis presents a behavior where the average response time converges to nearly 2 milliseconds when using both weak or strong data consistency levels. It means that it can have the same performance for writing operations independently of the data consistency level chosen when the number of concurrent requests is high. This may indicate that the number of concurrent users executed is not sufficient to evaluate Redis' limits in this aspect. As an in-memory database, its memory usage was not significantly affected (see Table 5).

Response time in reading operations. MongoDB outperforms Cassandra in all scenarios. While MongoDB presents a maximum average response of nearly 60 ms, Cassandra presents more than 4 times this value, with a maximum average response time of around 250 ms. When comparing the adoption of weak or strong data consistency levels, MongoDB has a similar growth for both levels as the number of concurrent users (and consequently, requests) grows. On the other hand, Cassandra has more problems in providing a faster response time when the strong data consistency level is adopted, as the response time grows linearly with the number of concurrent users. Besides, Cassandra shows a significant 183% increase (from 91 to 258) in the scenario with 10 concurrent users, while MongoDB has the lowest impact in the same scenario, with an increase of around 9%. These results show that adopting a strong data consistency level in Cassandra strongly impacts reading operations in a negative way.

Performance summary. The operational mechanisms of each DBMS help explain the experimental results. For read operations, Cassandra's in-memory structure enhances request performance under weak consistency. However, the need to communicate with other nodes introduces network latency under strong consistency, leading to significant performance degradation. On the other hand, MongoDB experiences inherent latency from disk access, resulting in poorer performance under

weak consistency, but it does not show as much variation when verifying read data across replica nodes in stronger consistency modes. In write operations, Cassandra's requirement to persist changes to disk means that the performance gain from memory in weak consistency is limited, resulting in a less significant impact when transitioning to strong consistency. In contrast, MongoDB's primary node must validate that writes are acknowledged by all replica set members, leading to a more pronounced performance decline when changing consistency levels.

Multi-regions deployment

We also executed the experiments in multiple cloud regions. In these experiments, the primary database node must communicate with replica nodes geographically separated by large distances. As in the previous subsection, we first analyze the results for throughput (Fig. 4) and then, response time (Fig. 5) illustrates how the different scenarios affect the different DBMSs.

Throughput in writing operations. The results indicate all DBMSs are highly affected by consistency switching (from weak to strong). For writes, all DBMSs handles over 90% less operations per second in all concurrent user scenarios. Cassandra presents a 98% decrease (from around 1300 to around 25) with 2 concurrent users, as it is capable to handle 1300 requests per second with weak consistency level and 25 with strong consistency level. In their worst-case scenarios, MongoDB shows 94% decrease (from around 475 requests per second to around 26) with 2 users and Redis shows 95% decrease (from around 1580 to around 70) with 3 users.

Throughput in reading operations. For readings, although MongoDB is the most affected, both Cassandra and MongoDB present similar performance degradation when strong consistency levels is set. In its worst case, MongoDB handles over 89% less operations (from around 42 to around 4) in the 1-user scenario, while the worst-case Cassandra results show a 66% decrease (from around 23 to around 7) in requests per second, considering the same 1-user scenario.

Response time in writing operations. Considering response time results, it is noticeable that writes

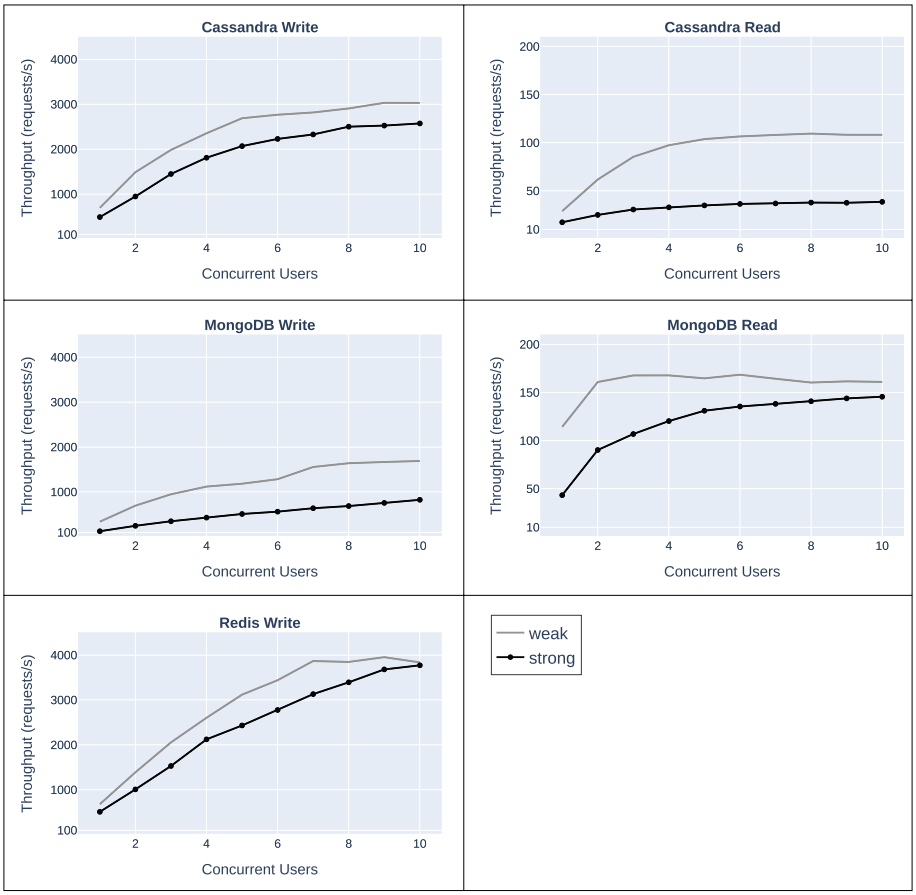

Fig. 4 Throughput results (in requests/s) for writing and reading operations of each NoSQL DBMS analyzed in multiple regions deployment. Scales differ on the y-axis for readability

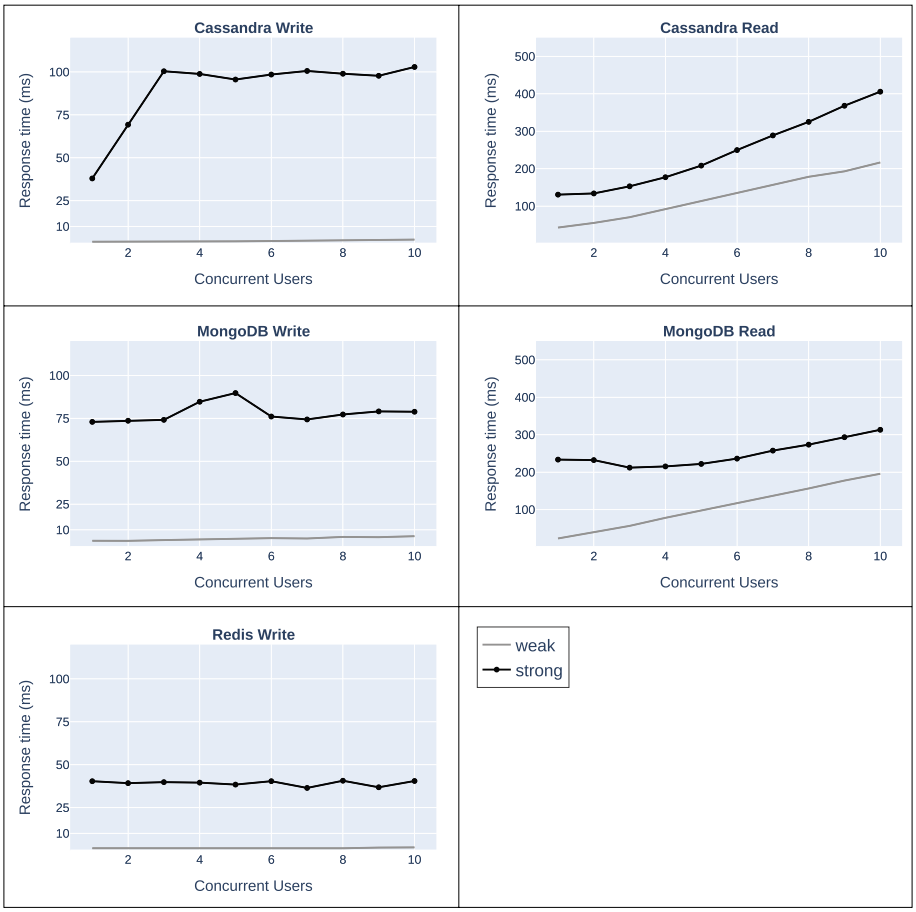

Fig. 5 Response time results (in ms) for writing and reading operations of each NoSQL DBMS analyzed in multiple regions deployment. Scales differ on the y-axis for readability

present the most significant increase in response time. All DBMSs take over 1,000% more time to process requests due to the need to validate data across the multi-region nodes. Cassandra exhibits the highest increase, taking around 9,000% (from around 1 to around 100 ms) more time to respond with 3 concurrent users. MongoDB, in its worst case, presents a 1,936% increase (from around 4 to around 74 ms) with 2 users. Redis shows a 3,122% increase (from around 1.2 to around 39 ms) with 3 users. Besides the single-region performance analysis, where Redis was not significantly affected, the metrics in this case are highly impacted by consistency switching. This may indicate that the main reason for performance variation is not necessarily the load but the latency among the nodes.

Response time in reading operations. For readings, both Cassandra and MongoDB exhibit the most significant impact with 1 user. In this case, Cassandra takes 205% more time to process reads: with weak consistency, it takes an average of 42 ms, while with strong consistency, it takes 130 ms. MongoDB, under the same conditions, shows an over 900% increase in read time (from around 22 ms to around 232 ms). On average, MongoDB's response time is the most affected when a strong consistency level is set. Since reads are low-cost operations for NoSQL databases and are usually processed quickly, the latency involved in communication among the nodes has a significant impact on the system's overall performance.

Performance summary. In a multi-region deployment, Cassandra is more affected by write operations than by read operations due to its data sharding mechanism, which divides the dataset into chunks and distributes them across multiple nodes. During reads, this mechanism allows for parallel validation of replicated data, as each node validates its own data chunks. However, in write operations, the system experiences greater performance loss because it must allocate incoming data to the appropriate chunks and generate replicas for other nodes. Conversely, MongoDB shows a more pronounced decline in read performance because it requires coordination among all replicas at

the primary node, resulting in larger data packets and increased latency. Redis, while consistent in performance within a single region, is significantly impacted in multi-region deployments, where its straightforward mechanism of waiting for replica communication is hindered by the increased latency between nodes.

Resources usage analysis

To analyze the resource usage of the VMs in each scenario executed in the experiment, we used a monitoring agent from GCP. Table 5 presents the average CPU and memory consumption for each scenario analyzed. The data was collected every minute during all scenarios for the primary DB node. For each group of parameters in the table, the resource usage was summarized across workload variations from 1 to 10 concurrent users, and their average values were extracted.

Table 5 Resource usage for all scenarios evaluated. The CPU and Memory usage values are the average of the scenarios with different numbers of concurrent users

Deployment	DBMS	Consistency	Operation	CPU	Memory
Single-region	Cassandra	Weak	Read	57,43%	43,46%
		Weak	Write	60,10%	39,17%
		Strong	Read	66,64%	43,75%
		Strong	Write	56,74%	43,29%
	MongoDB	Weak	Read	15,63%	41,47%
		Weak	Write	59,80%	20,15%
		Strong	Read	32,52%	40,99%
		Strong	Write	50,68%	42,87%
	Redis	Weak	Write	27,06%	13,12%
		Strong	Write	26,21%	13,17%
Multi-region	Cassandra	Weak	Read	7,65%	41,40%
		Weak	Write	47,31%	22,55%
		Strong	Read	9,94%	41,45%
		Strong	Write	9,12%	43,23%
	MongoDB	Weak	Read	7,65%	41,34%
		Weak	Write	48,43%	20,07%
		Strong	Read	9,71%	41,43%
		Strong	Write	9,12%	43,43%
	Redis	Weak	Write	29,61%	14,88%
		Strong	Write	4,43%	15,27%

We can observe that memory usage remains relatively stable across all scenarios, regardless of the database or configuration used, whereas CPU usage shows significant variation depending on the consistency level and deployment type. The VMs running Cassandra, MongoDB, and Redis consistently stayed below 70% CPU and memory usage, ensuring there were no resource bottlenecks. In addition, multi-region deployments with stronger data consistency levels exhibited lower CPU usage compared to single-region deployments. This suggests that, while requests take longer in a stronger consistency setting due to additional validations, these validations do not place a significant burden on CPU resources while waiting for responses from replicas.

Additionally, we analyzed the resource usage of the client VM (*client-node*), which is responsible for running the workload script and generating requests for the various scenarios. This analysis was conducted to ensure that the workload generation was not constrained by resource limitations (see Fig. 6). Despite using a VM with more available resources, the CPU and memory consumption did not exceed 30%, confirming that the workload generation ran efficiently without being impacted by hardware limitations in any of the scenarios tested.

Statistical analysis

Finally, we examined whether the throughput and response time results differed significantly when using weak versus strong data consistency levels for each

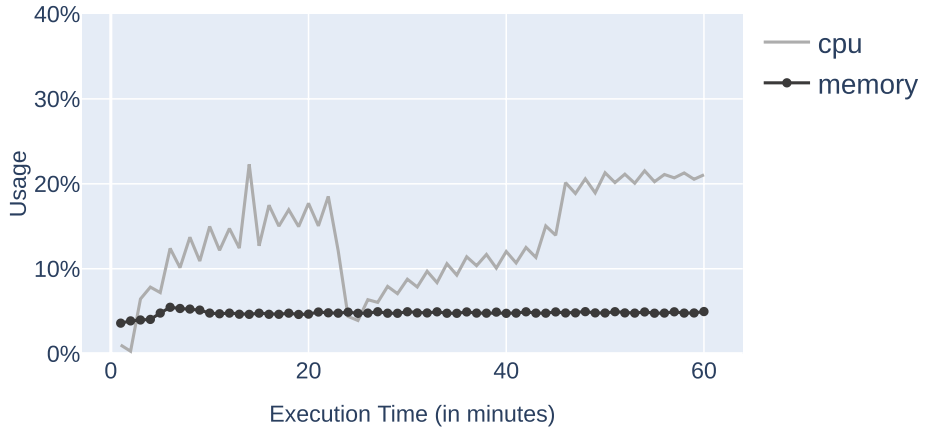

Fig. 6 client-node VM resource usage

DBMS and operation. We conducted statistical tests to determine whether adopting a weak or strong data consistency level resulted in statistically significant differences. To determine whether a parametric or non-parametric test should be used on the collected data, we first assessed whether the data followed a normal distribution using D'Agostino and Pearson's test [35]. For all cases, we considered a confidence interval of 95%. We utilized the Student's T-Test [36] for data that followed a normal distribution and employed the Mann-Whitney U Test [37] for data that did not follow a normal distribution.

The results obtained are presented in Tables 6 and 7 for the single- and multi-region, respectively. The p-value, highlighted in bold when below 0.05, indicates a statistical difference between adopting the weak and strong data consistency levels. For the single-region deployment, it indicates a significant difference in 3 out of 5 scenarios when considering each throughput (TP) and average response time (RT) value. However, it indicates no statistically significant variation for writing operations in Cassandra and reading operations in MongoDB. In the multi-region deployment, all five scenarios considered present significant variation when both weak and strong data consistency levels are compared.

Table 6 Statistical test results for single-region deployment

Database	Operation	TP data normality	TP p-value	RT data normality	RT p-value
Cassandra	Read	False	**0.001706**	True	**0.000316**
Cassandra	Write	True	0.163083	True	0.052704
MongoDB	Read	False	**0.001315**	True	0.264324
MongoDB	Write	True	**0.000374**	True	**4.906e-09**
Redis	Write	True	0.398822	False	**0.025748**

Table 7 Statistical test results for multi-region deployment

Database	Operation	TP data normality	TP p-value	RT data normality	RT p-value
Cassandra	Read	False	**0.001008**	True	**4.741e-03**
Cassandra	Write	True	**0.000000**	False	**1.827e-04**
MongoDB	Read	False	**0.000183**	True	**4.113e-06**
MongoDB	Write	True	**0.000002**	True	**1.799e-19**
Redis	Write	True	**0.000007**	False	**1.827e-04**

Findings and discussion

This section synthesizes our findings across all evaluated metrics. We analyze how the DBMSs behaved under the evaluated environment, particularly their response to different consistency levels and workloads. Table 8 illustrates the performance difference between these consistency levels, showing the percentage of increasing or dropping for each performance metric analyzed considering the deployment types proposed.

Performance in writing operations. The results indicate that as the number of concurrent users increases, all DBMSs can handle more requests simultaneously, as expected. However, despite the growth in the number of concurrent users, there is a consistent decrease in throughput and, consequently, an increase in response time for all DBMSs when transitioning from the weak to the strong data consistency level. This trend becomes more pronounced in multi-region deployments, where the difference in all writing operation metrics increases almost tenfold compared to single-region deployments. Specifically, MongoDB exhibits the most significant increase in single-region deployment (+175% response time and -58% throughput), while Cassandra is the most impacted DBMS in multi-region deployment (+6,028.34% response time and -97.54% throughput). Redis outperforms the other DBMSs in writing performance in single-region deployment, while MongoDB excels in multi-region deployment.

Performance in reading operations. Since Redis lacks support for data consistency levels in reading operations, our evaluation primarily centered around Cassandra and MongoDB. In the single-region deployment, Cassandra is more impacted by the data consistency level switch, presenting an increase of 175% in the response time and a decrease of 62% in the throughput. In the multi-region scenario, MongoDB presents the worst performance, losing around 57% of throughput and increasing around 237% of its average response time. While Cassandra demonstrated a superior average response time in reading

Table 8 Comparison of metric variances in weak-to-strong data consistency level transition. The DBMSs with the largest difference in each deployment, metric, and operation are marked in bold. The metrics analyzed are throughput (TP) and response time (RT)

		Read		Write	
		TP	RT	TP	RT
Single region	Cassandra	**-62.30%**	**+175.15%**	-22.20%	+38.79%
	MongoDB	-25.92%	+46.56%	**-58.89%**	**+175.64%**
	Redis			-17.72%	+20.55%
Multi-region	Cassandra	-50.24%	+106.93%	**-97.54%**	**+6,028.34%**
	MongoDB	**-57.88%**	**+237.86%**	-92.99%	+1,560.34%
	Redis			-94.83%	+2,774.35%
Average	Single region	-44.11%	+110.86%	-32.94%	+78.33%
	Multi-region	-54.06%	+172.40%	-95.12%	+3,454.34%

operations in the multi-region deployment compared to the single-region, the disparity between weak and strong data consistency metrics in the multi-region deployment was less pronounced than in the single-region.

Resource usage. The monitoring results revealed that different data consistency levels on DBMSs are not necessarily resource-intensive. Instead, it depends on how each system manages additional validations for replication synchronization, especially in stronger data consistency settings. Furthermore, the results indicated that memory usage remains almost constant across varying concurrent user workloads, suggesting that the initial memory allocation upon DBMS startup generally contains sufficient resources for system operation. However, it is notable that CPU usage exhibits the most significant variation in consumption. It is particularly noticeable in reading operations with Cassandra, where it experiences the most pronounced increase across concurrent user workloads. A similar trend was observed in Redis' writing operations.

Total data size. The amount of data involved in the test execution depends on the number of requests handled by DBMSs during each run. Since the test is time-limited, a system's ability to handle more requests per second (throughput) results in more data being stored or retrieved. As defined in "Database configurations" section, writing operations in the experiments insert a single row of data with 8,192 bytes, while read operations request a slice of 1,000 rows. Under these conditions, we calculate the average data size stored and retrieved per execution. For reading operations, a total of 230.41 GB was recovered considering the experiments in single-region deployment, while in multi-region deployments, this number was 76.03 GB. When executing writing operations, 4.31 GB was stored during the experiments in the single-region deployment and 2.35 GB during the experiments in the multi-region deployment. Since the DBMSs require more time to handle an operation in the multi-region deployment, this configuration had fewer operations being completed within the pre-defined experiment duration time and, consequently, fewer data being processed.

Limited cloud services. Although our extensive experiments allowed us to get valuable insights and contribute to understanding the performance and behavior of the evaluated DBMSs, the experiments were conducted on a restricted free-tier Google Cloud Platform, which imposed limitations on our experiments. For example, the allocated number of vCPUs was limited, restricting us from establishing a testbed with better resources on each VM and employing a higher workload. Additionally, it constrained us from instantiating more nodes for each evaluated DBMS and testing other configurations of consistency or replica management, such as Redis Cluster [38].

Unique target host. In our experiments, we designated one node (i.e., the *primary-db-node-a/b*) to manage the requests, effectively serving as the coordinator consistently. This allowed us to have a fair comparison, given that MongoDB and Redis adopt a primary-secondary architecture where the primary node inherently serves as the request coordinator. As mentioned previously, Cassandra has a masterless architecture, allowing any node in the cluster to handle incoming requests and perform identical tasks to other nodes. In future work, we aim to investigate the performance of Cassandra using this masterless configuration.

Conclusions and future work

This work explored the impact of data consistency levels on the performance of three popular NoSQL DBs, namely Cassandra, MongoDB, and Redis, in different cloud-based environments. Our findings indicate that enhancing consistency among nodes leads to a considerable decline in performance, particularly under heavier workloads, resulting in a reduced number of requests handled per second (throughput). Besides, the response time variation is not notable in scenarios with few concurrent requests, but it can quickly escalate when the number of concurrent requests grows. Our experiments in single- and multi-cloud regions reveal that the geographical disposition of nodes significantly impacts the time required for the system to ensure data consistency between replicas. Furthermore, our experiments revealed statistically significant differences in throughput and response time for writing and reading operations when adopting different data consistency levels. These results offer valuable insights into the impact of data consistency levels on the performance of these NoSQL DBMSs and can assist in selecting the best DB system for specific use cases. In future research, we plan to analyze the performance impacts in Cassandra when using the masterless structure and incorporating a higher number of VMs. Additionally, we will utilize the YCSB benchmark [39] alongside Locust to enhance our performance analysis.

Abbreviations

AWS	Amazon Web Services
DB	Database
DBMS	Database management system
GCP	Google Cloud Platform
NoSQL	Not-only SQL
RT	Response time
SQL	Structured query language
TP	Throughput
VM	Virtual machine
YCSB	Yahoo! Cloud Serving Benchmark

Acknowledgements
The authors declare there is no additional authors or involved people in this work.

Authors' contributions
S.F.: Methodology, Experiment setup and execution, Data processing, Generation of resources (tables, charts and images) and Writing (original draft). J.M. and E.A.: Writing (review and editing) and Supervision. B.N. and W.T.: Writing (review and discussion).

Funding
The authors declare that they did not receive any funding to produce this work.

Data availability
The data presented in "Experiments results and discussion" section are available at Google Drive in raw CSV (comma-separated values) format without processing and can be accessed via the following link: https://drive.google.com/drive/folders/154JvVfBWL6qeJ3K0Bs6eobPJPSjlCzfc?usp=sharing.

Declarations

Competing interests
The authors declare no competing interests.

Received: 20 July 2024 Accepted: 20 October 2024
Published: 27 November 2024

References

1. Ab Rashid Dar RD (2016) Survey on scalability in cloud environment. Int J Adv Res Comput Eng Technol 5(7):2124–2128
2. Nadiminti K, De Assunçao MD, Buyya R (2006) Distributed systems and recent innovations: Challenges and benefits. InfoNet Mag 16(3):1–5
3. Abualkishik AZ, Alwan AA, Gulzar Y (2020) Disaster recovery in cloud computing systems: An overview. Int J Adv Comput Sci Appl 11(9):702–710
4. Ledmi A, Bendjenna H, Hemam SM (2018) Fault tolerance in distributed systems: A survey. In: 2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS), IEEE, pp 1–5
5. Abd Alnabe N, Zeebaree SR (2024) Distributed systems for real-time computing in cloud environment: A review of low-latency and time sensitive applications. Indones J Comput Sci 13(2):2549–7286
6. Mansouri Y, Prokhorenko V, Babar MA (2020) An automated implementation of hybrid cloud for performance evaluation of distributed databases. J Netw Comput Appl 167(102):740
7. Al Shehri W (2013) Cloud database database as a service. Int J Database Manag Syst 5(2):1
8. Shapiro M, Sutra P (2018) Database consistency models. arXiv preprint arXiv:1804.00914
9. Gorbenko A, Romanovsky A, Tarasyuk O (2020) Interplaying cassandra nosql consistency and performance: A benchmarking approach. In: Dependable Computing-EDCC 2020 Workshops: AI4RAILS, DREAMS, DSOGRI, SERENE 2020, Munich, Germany, September 7, 2020, Proceedings 16, Springer, pp 168–184
10. Gomes C, de O Junior MN, Nogueira B, Maciel P, Tavares E (2023) Nosql-based storage systems: influence of consistency on performance, availability and energy consumption. J Supercomput 79(18):21424–21448
11. Wada H, Fekete AD, Zhao L, Lee K, Liu A (2011) Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. CIDR 11:134–143
12. Diogo M, Cabral B, Bernardino J (2019) Consistency models of nosql databases. Futur Internet 11(2):43
13. Strauch C, Sites ULS, Kriha W (2011) Nosql databases. Lect Notes Stuttgart Media Univ 20(24):79
14. Moniruzzaman A, Hossain SA (2013) Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. arXiv preprint arXiv:1307.0191
15. DB-Engines (2024) DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed 20 Jan 2024
16. Brewer EA (2000) Towards robust distributed systems. PODC, Portland, OR 7:343477–343502
17. Gilbert S, Lynch N (2012) Perspectives on the cap theorem. Computer 45(2):30–36
18. Hewitt E (2010) Cassandra: the definitive guide. O'Reilly Media Inc, Newton
19. Membrey P, Plugge E, Hawkins T, Hawkins D (2010) The definitive guide to MongoDB: the noSQL database for cloud and desktop computing. Springer, New York
20. Sanfilippo S, Noordhuis P (2009) Redis. https://redis.io. Accessed 10 June 2024
21. Chen S, Tang X, Wang H, Zhao H, Guo M (2016) Towards scalable and reliable in-memory storage system: A case study with redis. In: 2016 IEEE Trustcom/BigDataSE/ISPA, IEEE, pp 1660–1667
22. Han J, Haihong E, Le G, Du J (2011) Survey on nosql database. In: 2011 6th international conference on pervasive computing and applications, IEEE, pp 363–366
23. Mohamed MA, Altrafi OG, Ismail MO (2014) Relational vs. nosql databases: A survey. Int J Comput Inf Technol 3(03):598–601
24. Khan W, Kumar T, Zhang C, Raj K, Roy AM, Luo B (2023) Sql and nosql database software architecture performance analysis and assessments—a systematic literature review. Big Data Cogn Comput 7(2):97
25. Abu Kausar M, Nasar M, Soosaimanickam A (2022) A study of performance and comparison of nosql databases: Mongodb, cassandra, and redis using ycsb. Indian J Sci Technol 15(31):1532–1540
26. Gandini A, Gribaudo M, Knottenbelt WJ, Osman R, Piazzolla P (2014) Performance evaluation of nosql databases. In: Computer Performance Engineering: 11th European Workshop, EPEW 2014, Florence, Italy, September 11-12, 2014. Proceedings 11, Springer, pp 16–29
27. Abramova V, Bernardino J, Furtado P (2014) Which nosql database? a performance overview. Open J Databases (OJDB) 1(2):17–24
28. Wang H, Li J, Zhang H, Zhou Y (2014) Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra. In: Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware, Springer, pp 71–82
29. Gomes C, Borba E, Tavares E, Junior MNdO (2019) Performability model for assessing nosql dbms consistency. In: 2019 IEEE International Systems Conference (SysCon), IEEE, pp 1–6
30. Haughian G, Osman R, Knottenbelt WJ (2016) Benchmarking replication in cassandra and mongodb nosql datastores. In: International Conference on Database and Expert Systems Applications, Springer, pp 152–166
31. Ferreira S, Andrade E, Mendonça J (2021) Uma abordagem experimental para avaliar os níveis de consistência do banco de dados nosql cassandra. In: Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho, SBC, pp 156–167
32. Heyman J, Byström C, Hamrén J, Heyman H (2012) Locust.io. https://locust.io/. Accessed 10 June 2024
33. Pradeep S, Sharma YK (2019) A pragmatic evaluation of stress and performance testing technologies for web based applications. In: 2019 Amity International Conference on Artificial Intelligence (AICAI), IEEE, pp 399–403
34. Montgomery DC (2017) Design and analysis of experiments. John wiley & sons, Hoboken
35. DIAgostino R (1971) An omnibus test of normality for moderate and large sample sizes. Biometrika 58(34):1–348
36. Kotz S, Johnson NL (eds) (1992) The Probable Error of a Mean, Springer New York, New York, pp 33–57. https://doi.org/10.1007/978-1-4612-4380-9_4
37. McKnight PE, Najab J (2010) Mann-whitney u test. The Corsini encyclopedia of psychology, Wiley, Hoboken, New Jersey, p 1
38. Redis (2024) Redis Documentation. https://redis.io/docs/latest/. Accessed 17 June 2024
39. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing, Association for Computing Machinery, New York, New York, pp 143–154

Publisher's Note

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com

APPENDIX B – Experimental Performance Analysis of Data Consistency Levels in NoSQL Databases

Ferreira et al. «Experimental Performance Analysis of Data Consistency Levels in NoSQL Databases». In *Software: Practice and Experience*. 2025.

Author's contribution: The author was responsible for developing methodology, experiment setup and execution, data processing, resource generation (including tables, charts, and images), and writing the original draft.

Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

RESEARCH ARCTICLE

Experimental Performance Analysis of Data Consistency Levels in NoSQL Databases

Saulo Ferreira*[1] | Júlio Mendonça[2] | Ermeson Andrade[1]

[1]Universidade Federal Rural de Pernambuco, Pernambuco, Brazil
[2]Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg

Correspondence
*Saulo Ferreira, Rua Dom Manuel de Medeiros, Recife, Pernambuco, Brazil.
Email: saulo.gomesferreira@ufrpe.br

Abstract

NoSQL database management systems (DBMSs) are designed to handle large-scale data for modern applications. These systems often operate in a distributed manner, allowing data to be spread across multiple nodes to ensure replication, reduce data loss, and facilitate recovery. The consistency level in these DBMSs determines how synchronized data is across nodes, influencing the trade-off among consistency, availability, and system performance. Given that different applications have unique requirements, understanding the impact of various consistency levels is essential. This study conducts an in-depth analysis of how consistency level choices affect the performance of three leading NoSQL DBMSs: Cassandra, MongoDB, and Redis. These systems were evaluated under different consistency configurations, user loads, and workloads, with performance metrics including average response time and operations per second. Our results show that Cassandra and Redis handle write operations faster than MongoDB, though Cassandra experiences significant slowdowns of up to 200% when switching to strong consistency. This performance degradation is observed for both read and write operations, making Cassandra the most affected DBMS when opting for strong consistency. The detailed findings offer valuable insights into the trade-offs between performance and consistency in these DBMSs, providing guidance for database engineers in selecting appropriate consistency levels based on their application needs.

KEYWORDS:
Consistency; Databases; NoSQL; Performance

1 | INTRODUCTION

Distributed systems enable multiple machines (referred to as nodes) to collaborate and communicate over a network[1]. These systems necessitate strong and efficient communication to ensure quick responses and resilience against system failures[2]. When utilizing distributed systems for workload sharing and enhanced performance through parallel processing, low-latency network communication is essential for efficient task orchestration and division. However, distributed database (DB) systems do not inherently require low-latency communication for data storage and replication within a DB cluster. In such systems, it is crucial for nodes to maintain up-to-date data by communicating with replica nodes to persist and validate changes[3].

When adopting distributed NoSQL (Not Only SQL) database systems, strategies for enhancing database operation performance, such as reading and writing, often emerge in response to communication failures and delays. One such strategy is

asynchronous replication, where distributed NoSQL database systems process incoming requests exclusively on the primary node and then propagate changes to other nodes in the cluster in the background[4]. While this approach can reduce network communication and provide quick responses for applications, it may not be suitable for all scenarios, as the database management system does not guarantee data consistency across all nodes. In the event of a failure in the primary node, there is no assurance that a secondary node can take its place with up-to-date data. Therefore, to ensure data consistency among multiple nodes, NoSQL DBMSs typically offer various consistency-level configurations. However, these strategies can lead to performance degradation[5].

While some applications require the most robust consistency approach (e.g., banking applications) to ensure all data will be up-to-date among all nodes in the DB cluster, even if this condition means performance loss, other applications (e.g., social networks) may afford a weak consistency level to deliver better performance to its users. In this context, a key challenge is still fully comprehending the trade-offs between performance and consistency levels to deploy distributed NoSQL DB systems adequately. Several studies have investigated consistency levels in database systems. For example, Bailis et al.[6] analyze the trade-offs involved in consistency approaches using probabilistic models to evaluate the frequency with which an eventually consistent DBMS returns inconsistent data. Nevertheless, most research has focused on how stronger consistency levels impact average response time[7], with fewer studies analyzing multiple DBMSs simultaneously. Diogo et al.[8] examined the consistency strategies of various NoSQL DBMSs by comparing the time required for each system to regain consistency after an operation, while Haughian et al.[9] employed a benchmarking strategy to compare MongoDB and Cassandra across different consistency configurations. Still, there remains a gap in these works regarding the analysis of important performance metrics, such as response time and throughput, under the stress of varying workloads.

Therefore, this work experimentally evaluates performance metrics of NoSQL DBMS using distinct data consistency levels. To achieve this, a full factorial experimental design is applied to analyze the influence of various scenarios on response time and throughput in greater detail. We selected three widely adopted NoSQL DB systems for our analysis, based on the DB-Engines Ranking[10]: MongoDB, Redis, and Apache Cassandra. The experimental results demonstrated that varying the data consistency level had a minimal impact on write operations for Redis and read operations for MongoDB. In contrast, Cassandra experienced the most significant performance degradation in both write and read operations. We summarize the main contributions of this work as follows:

- We analyze how the performance of different DBMS is affected by different configurations for ensuring data consistency.
- We identify the best and worst scenarios that different DBMSs can present for response time and throughput under various data consistency levels and workload conditions.
- We statistically analyze whether the throughput and response time measurements of writing and reading operations are significant or not.
- Our findings provide evidence for the use of NoSQL DBMS regarding data consistency levels and the performance trade-offs in terms of response time and throughput.

The remainder of the paper is organized as follows. Section 2 presents related works. Section 3 explains the fundamental concepts adopted. Section 4 describes the experimental approach adopted and the testbed used for the experiments. Section 5 discusses the achieved results. Finally, Section 6 concludes the paper and outlines future objectives.

2 | RELATED WORKS

Several works have investigated various aspects of NoSQL DBMS. In this section, we present studies related to the evaluation of different features of NoSQL systems, including data consistency levels, and highlight the main research contributions of our paper. Gomes et al[11] developed an approach based on Petri nets to evaluate the consistency and availability of a three-node Cassandra cluster. That work revealed that the number of replicas and the response time of the coordinator node are the main reasons for decreasing the database performance. Similarly, Gomes et al[12] proposed using Petri nets to evaluate the database availability and consistency of cloud-based NoSQL databases. Their work found that increasing consistency levels improves the ability to access the newest data by 91.4%, but it also revealed that the annual unavailability time can increase to more than 18 hours. Araújo et al[13] proposed a similar approach employing Petri nets and demonstrated that a strong consistency level

Work	Evaluate consistency level	Compare multiple DBs	Apply several workloads	Performance Metrics Analyzed
Wang et al[15]	Yes	Yes	No	No
Gomes et al[11]	Yes	No	No	*TP & **RT
Gomes et al[12]	Yes	No	No	RT
Araújo et al[13]	Yes	No	No	RT
Tang and Fan[14]	No	Yes, 5	Yes	TP
Gorbenko et al[7]	Yes	No	Yes	TP & RT
Ferreira et al[16]	Yes	No	Yes	RT
Haughian et al[9]	Yes	Yes, 2	Yes	TP
This work	**Yes**	**Yes, 3**	**Yes**	**TP & RT**

TABLE 1 Comparison of related works main characteristics. *TP = Throughput, **RT = Response Time.

results in the system returning inconsistent data in only 1.4 out of 100,000 read requests, whereas the weak consistency level may escalate this number to 2,534 per 100,000.

In a different approach, Tang and Fan[14] conducted a benchmarking evaluation comparing the throughput results of five NoSQL databases, and the results indicated Redis as having the best overall performance, while MongoDB exhibited the worst performance. Wang et al[15] compared the DBMS Cassandra and HBase considering a different number of replicas and consistency levels. They also conducted comprehensive stress benchmarks to evaluate the performance of each DB when handling read and write operation requests under heavy loads.

Gorbenko et al[7] proposed a benchmark evaluation of different Cassandra's consistency levels deployed on AWS EC2. The main objective of that work was to study how different consistency levels affect Cassandra's performance through high workloads. It revealed that strong consistency can decrease performance by 25%, and the biggest impact is on read operations. Additionally, they proposed a set of optimized consistency settings for Cassandra that can maintain strong consistency while maximizing performance. In a similar approach, Ferreira et al[16] investigated the consistency of Cassandra and analyzed the associated latency across different consistency levels. The results revealed a growth superior to 100% in the latency (from the weakest to the strongest consistency level) under different workloads.

Using a comprehensive approach, Haughian et al[9] analyzed the throughput of Cassandra and MongoDB across various consistency levels, observing changes in the number of operations per second as the number of nodes required for coordinator operation varied. The study's findings revealed that MongoDB's master-slave configuration is effective in reducing replication impacts, with writing operations being more significantly impacted at higher workloads. However, in a non-replication scenario, the authors showed that Cassandra can scale better than MongoDB.

Unlike previous studies, this work aims to evaluate a multinode cluster deployed on local containers for three different database systems: Cassandra, MongoDB, and Redis. Our experiments employ a full factorial design to generate all possible combinations of configurations. Table 1 presents a comparative analysis of the main characteristics of related works and this paper, highlighting the significance of our findings within the context of previous research. This study has the potential to aid in the configuration and implementation of NoSQL databases across various industries and applications. By generating detailed data on the performance and scalability of Cassandra, MongoDB, and Redis under different configurations, we provide valuable insights for organizations seeking to optimize their database environments.

3 | BACKGROUND

This section presents the main concepts of the paper. We first describe the CAP Theorem and how distributed systems are designed to deliver two out of the three primary properties. Next, we explain NoSQL database management systems, focusing on three specific systems adopted in this paper: Cassandra, MongoDB, and Redis.

3.1 | CAP Theorem

Distributed databases have fundamental quality constraints, as described by Brewer's CAP Theorem [17]. Brewer's theorem states that a distributed database system can deliver only two of three desired characteristics: consistency, availability, and partition tolerance. Consistency describes the system property that prioritizes validating synchronously all primary and secondary nodes in order to guarantee the data is always being updated equally for all replicas. Availability refers to the ability of a system to remain accessible and responsive to client requests at all times, even in the face of hardware or software failures, network disruptions, or other unexpected events. Partition tolerance describes a system that can recover from failures on its nodes because the other nodes have enough information and capability to replace them. Note that when designing a distributed system under the possibility of network partitions, it is necessary to prioritize either availability or data consistency, as both cannot be achieved simultaneously in such cases. Designers must balance the trade-offs and make informed decisions to meet user needs while addressing system limitations [18].

To choose availability over consistency, database systems usually adopt the eventual consistency paradigm, where the operation is updated only on the requested node and persisted asynchronously on other nodes. Thus, an arbitrary node may not have the latest data, but it will be updated at some point. This makes the system respond quickly to operations and does not block the system's transaction flow from other requests. However, some systems may require the most recent data, even though this type of need increases response time.

Alternatively, for applications where consistency is critical, analyzing data consistency in database systems through a synchronous replication mechanism becomes essential. The PACELC theorem was introduced by Abadi et al. [5] and highlights the trade-offs between latency and consistency, even in systems that are not partitioned but involve data replication. Although many DBMSs implement synchronous replication to ensure consistency by requiring each replica node to confirm an operation before committing to data updates, others may prioritize performance and availability, opting for asynchronous replication by default to reduce operation times. When synchronous replication is employed, it guarantees that all replicas store the same data, with all transactions persisted simultaneously. The synchronization of data between nodes significantly impacts system performance and response time. For example, if one node fails, the system can ensure that other nodes can efficiently take over, maintaining the same data set with the latest updates. This ability to recover quickly depends on the effectiveness of data synchronization across all nodes.

3.2 | NoSQL Databases

NoSQL databases [19] are a type of storage system that does not adhere to a well-defined data structure and does not have rigid data dependencies. They are able to store data in various formats, such as key-value pairs, structured tables, document-oriented databases, and more [20]. NoSQL databases are generally used to store large amounts of data with efficient features to retrieve data quickly by indexing or mapping this data. They are commonly deployed in clusters, using several nodes to divide the information into separated physical machines. This makes the database system focus on mapping the data addresses and not concentrating the requests in a single point, consequently, dividing the responsibility among the other nodes and reducing the response time when the nodes are physically close to the users. In addition, deploying multiple nodes facilitates data replication, which can keep the system running even when a limited number of nodes fail.

3.2.1 | Cassandra

Apache Cassandra [21] is an open-source NoSQL database focused on performance, developed by Facebook and maintained by the Apache Foundation. Its data organization is based on wide-column tables, allowing for the definition of column names and types within a structure similar to relational databases but without data dependencies. As a result, Cassandra features a declarative language akin to SQL, which includes commands for insert, update, and select operations. It can be deployed across multiple nodes or data centers with a *masterless* architecture, enabling all nodes in a cluster to operate equivalently and act as coordinators for operations. To enhance performance, Cassandra utilizes an in-memory structure called Memtable to persist the most recent write operations for quick recovery. When the data in Memtable exceeds a preconfigured memory size threshold, it is moved to on-disk storage in the form of SSTables, where all indexes and table structure configurations are retained [22].

Although availability is one of Cassandra's main characteristics, it is possible to configure consistency levels to determine how each Cassandra node interacts with others when operations are requested. At higher consistency levels, the coordinator node waits for a greater number of nodes to confirm read or write operations on the database. To enhance performance in these

scenarios, Cassandra can write changes concurrently across multiple nodes. The main consistency levels in Cassandra are as follows:

- **ONE**. At this consistency level, the node handling the request (referred to as the coordinator) operates with minimal validation. For read operations, the coordinator node returns the data directly without verifying it with other nodes in the cluster. As a result, there is a risk of returning stale data if updates exist on other replicas. For write operations, the coordinator confirms the operation as soon as it is written to one replica, without waiting for replication to other nodes. This level prioritizes low latency and is well-suited for applications that can tolerate eventual consistency.

- **ALL**. This level enforces the highest consistency by ensuring that all replicas in the cluster are synchronized before an operation is confirmed. For read operations, the coordinator node queries every replica to verify that the data returned is the most up-to-date. For write operations, the coordinator waits for confirmation from all nodes in the cluster before acknowledging the operation as complete. While this provides strong consistency guarantees, it introduces higher latency and reduces throughput, as the slowest node in the cluster can delay the operation. This level is suitable for applications where strict consistency is critical, such as financial or transactional systems.

While Cassandra offers additional consistency levels, such as “QUORUM", we chose not to include them in order to focus specifically on the weakest and strongest consistency levels available for each database, aligning with the infrastructure used. Additionally, the decision to use “ALL" for both reads and writes may appear redundant because confirming write operations on all nodes means the read operation does not need to revalidate it. However, this choice is motivated by our objective to evaluate read and write operations separately through distinct test scenarios. We do not combine the operations in our experiments and instead divide them into individual test cases to assess the impact of each operation separately.

3.2.2 | MongoDB

MongoDB[23] is a document-based NoSQL database that organizes data in BSON, which is a JSON-like format. Although this database does not offer a declarative language similar to SQL, it provides its own command-line interface based on functions for inserting, modifying, and reading stored data. MongoDB employs MapReduce to accelerate data read operations. MapReduce is a method for grouping and filtering data to find documents based on conditions specified by users. Additionally, MongoDB uses an on-disk journal to propagate in-memory changes to data files on the primary node. However, it does not allow simultaneous updates of secondary replica nodes, which can make it costly at more robust consistency levels.

MongoDB is organized by default in a primary-secondary cluster structure, meaning it has a primary node that serves as the target for requests and orchestrates communication between the client and the database. The primary node performs operations and propagates changes to the secondary nodes, which can assume the primary role in the event of a primary node failure. Consequently, MongoDB uses an asynchronous mechanism to propagate data changes from primary nodes to secondary nodes, aiming to achieve fast response times in operations.

MongoDB also offers a consistent schema that determines how many nodes are expected to perform commit operations and validate read data[24]. It is possible to configure read and write concerns, which are parameters defined for each operation. The operation concern specifies how the operation will be committed, allowing for options such as monotonic writes or causal consistency. In this work, we adopt the *local* and *linearizable* configurations, as they represent the weakest and strongest consistency levels available for MongoDB. These configurations are defined as follows:

- **Local**: For read operations, the *local* configuration retrieves data from the local replica without guaranteeing that it is the most up-to-date version available across the cluster. For write operations, *w:1* ensures that the operation is validated only on the local node. This setting offers the weakest consistency but provides high performance and low latency, making it suitable for scenarios where eventual consistency is acceptable.

- **Linearizable**: For read operations, the *linearizable* configuration ensures that the data returned is the most recent and consistent version across all nodes. For write operations, *w:n* (where *n* is the number of replicas) ensures that the operation is validated by all or *n* nodes, providing the strongest data consistency level. While this configuration guarantees strict consistency, it comes with increased latency and reduced throughput due to the additional validation steps required.

3.2.3 | Redis

Redis is a multi-purpose, open-source database that can be utilized for various applications, including real-time social media analytics, ad targeting, and caching. As a result, Redis is often referred to as a data structure store, a broader term that reflects its diverse capabilities. Its data structure can handle multiple types of data and allows for data modification operations directly from the command-line interface. Redis runs entirely in memory, making it a popular choice for real-time responsive applications and reinforcing its ability to provide instant access to stored data[25].

While availability is the primary focus of Redis, it also offers functionality for configuring synchronous replication for certain operations. The Redis *WAIT* command enhances system consistency by ensuring that all replicas persist information to storage before the client receives a write commit. However, it is important to note that the *WAIT* command applies only to write operations, as Redis does not support consistency configurations for read operations. All read requests are always fetched from the master instance in memory.

4 | DISTRIBUTED NOSQL ENVIRONMENT

This section details the experimental setup adopted in this work, such as the physical environment, database configurations, workloads, and input parameters.

4.1 | The adopted testbed

The adopted testbed contains two different physical machines, the target and source machines (see Figure 1). The target machine was responsible for running the DBMS cluster environments for all DBMS used (i.e., Cassandra, MongoDB, and Redis). The source machine acted as a multi-thread client executing concurrent requests. The different DB environments were configured and run one at a time in the cluster. The target machine was configured with an Intel Core i7-1165G7 processor, having 8 cores of 2.80 GHz CPU and 32 GB RAM, running Ubuntu 20.04 operating system. The source machine was configured with an AMD Ryzen 5 4600G with 6 cores of 3.7GHz CPU and 16 GB RAM, running Windows 11 operating system. The target machine was configured with more RAM to be able to start multiple docker containers to simulate a DB cluster.

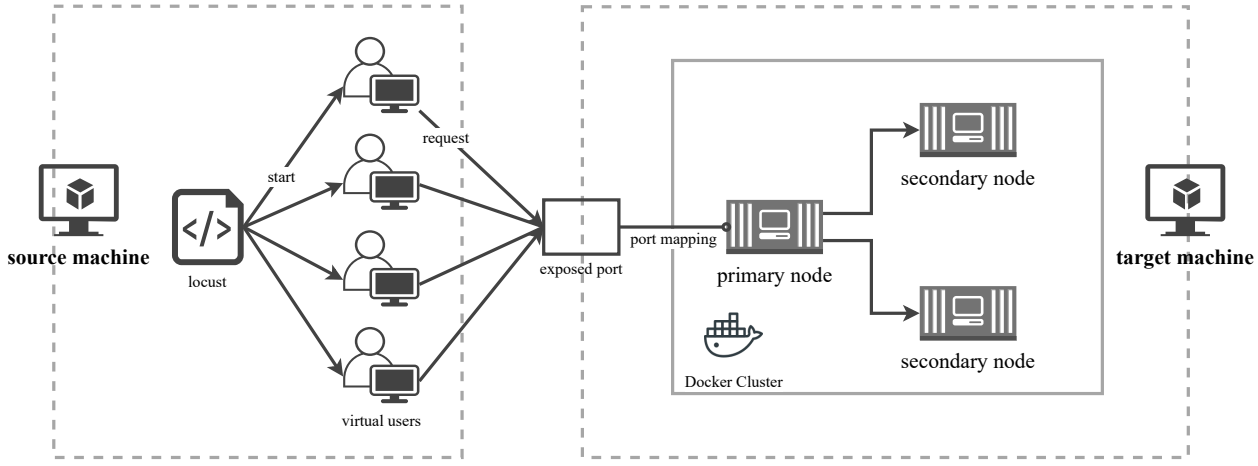

FIGURE 1 Implemented testbed.

We used the source machine to execute multi-threaded requests to the database cluster deployed on the target machine, acting as a workload generator. This setup allowed us to apply different levels of workload to the database cluster environments. To generate these varying workloads, we utilized Locust[26], version 2.12.1, an open-source load testing tool for Python. Locust

enabled us to run tests based on multiple settings, including the number of concurrent users, time intervals between requests, user spawn rates, and total duration.

The cluster environments were deployed on the target machine using three Docker containers, each provisioned with identical resources (1 vCPU and 4GB of RAM) and connected via a local network. These containers, referred to as nodes, shared a single 512 GB NVMe SSD with read speeds of 2500 MB/s and write speeds of 1500 MB/s. One of the containers was designated as the primary node, responsible for handling requests from the source machine, while the other two were configured as secondary nodes, receiving data updates from the primary node based on the chosen data consistency level. Following the setup of the testbed, each DBMS was configured to run under its weakest and strongest consistency settings to assess performance metrics. Additionally, we utilized Linux Traffic Control (TC)[27] to introduce network latency between the container nodes, addressing the limitation of running all nodes on a single machine. To simulate the conditions of multiple nodes in the same data center, we set the TC latency parameter to match the average latency measured between nodes in AWS's US East (North Virginia) region (the oldest AWS region with the most availability zones), which was approximately 5 milliseconds[28].

4.2 | Database Configurations

We selected the strongest and weakest consistency levels available for analysis in each DBMS. The objective was to maintain similar settings across all evaluated systems to ensure a balanced experiment.

For Cassandra, we deployed the official Docker Hub image version 4.1[1] and set it up across three nodes, with each node configured to replicate its data to the others. The weak consistency setting was defined as the consistency level ONE, and the strongest was defined as ALL. To read and write data on the Cassandra nodes, we created a table with 1 integer unique identifier and 10 string columns filled with the same data.

For MongoDB, we configured a three-node cluster with the official Docker image at version 7.0.12[2], where each node serves as both a primary and secondary node, replicating data bidirectionally with the other two nodes. The weakest consistency level for readings (read concern) was set as *local*, and the strongest consistency level was *linearizable*. The weakest consistency configuration for writing operations (write concern) was set to 1 (i.e., only one node is necessary to commit the operation), and the strongest was set to 3 (i.e., all the 3 nodes' confirmation is needed to commit the operation). The data format for reading and writing operations was created as a JSON-like document with only an integer identifier and 10 other key-value string pairs.

For Redis, we utilized the official Docker image, version 7.4[3]. While this DB does not offer full support for consistency-level configurations, it does provide synchronous replication as a means to achieve consistency. To use synchronous replication, it is necessary to use the WAIT command to specify the number of nodes that should confirm an operation. However, it is only possible for writing operations. Thus, we evaluated Redis only concerning them. For the weakest consistency level, we set the WAIT command to 1, and for the strongest, we set WAIT to 3. We used a data format that consisted of a single key with a unique value for each write operation, using the hash data type to store ten field-value pairs with string values.

For all databases, the data used was the same, following their respective input formats: a randomly generated unique identifier and a set of 10 predefined key-value or column-value data pairs. The reading operation was comprised of a select statement with no filters and a limit of 1000 rows or documents, while the write operation used an insert statement with the provided data. Furthermore, the TC tool is executed within the containers to introduce latency between instances. Whenever the container communicates with the specified IP, TC generates a delay in communication.

4.3 | Input Parameters and Evaluated Metrics

Table 2 presents the parameter configurations adopted in the experiments. These configurations correspond to the DBMS analyzed, the operations performed, and the workload applied. The first column is the number of concurrent users (or threads) making continuous requests to the database with no time interval between them. The second one corresponds to the consistency level applied. The third column is the operation performed (reading/writing), and the last one is the database evaluated. All of those configurations were used to generate a full-factorial[29] collection of scenarios with all possible combinations between

[1]Cassandra Docker image URI: https://hub.docker.com/layers/library/cassandra/latest/images/sha256-cb808e336f0a9258008ddab5d2d6caf054576d952014cac6bf4246e95f0d837f?context=explore

[2]MongoDB Docker image URI: https://hub.docker.com/layers/library/mongo/latest/images/sha256-6d245af0e8dffc8771f3e734b4ee0bc7fcde48fcffe3bf2664592dfda8e9e6a4?context=explore

[3]Redis Docker image URI: https://hub.docker.com/layers/library/redis/latest/images/sha256-bdaf8425515f443a223fd16d071c6903d42d8f3bee28ec0aa3b36ecace9afd80?context=explore

them. Each combination of those parameters resulted in an individual test scenario that was executed separately from the others. The total number of combinations is 100, considering 40 for MongoDB, 40 for Cassandra (varying 10 concurrency levels, 2 consistency levels, and 2 operations), and 20 for Redis (varying across 10 concurrency levels, 2 consistency levels, and using only 1 operation, as the consistency configuration is available only for write operations). Note that to generate requests based on the specified number of concurrent users, we employed the Locust configuration arguments detailed in Table 3 .

Concurrent users	Consistency level	Operation	Database
10-100	weak	read	Cassandra
(step of 10)	strong	write	MongoDB
			Redis

TABLE 2 Input parameter configurations.

During our analysis, we focused on two primary metrics to evaluate performance: throughput and average response time. While the throughput provides insights into the overall number of operations completed per second, the response time is a critical measure of latency variation resulting from changes in workload and consistency levels. We collect both metrics from the Locust output file for all scenarios tested[4].

5 | EXPERIMENTAL RESULTS AND DISCUSSION

This section presents and discusses the results of the experiments conducted in the different NoSQL DBMSs adopted using the testbed defined in Section 4. We focus on the discussion about consistency levels and how they impact the DB systems' operation.

5.1 | Throughput Analysis

Figure 2 presents the throughput of each DBMS in different scenarios. The y-axis indicates the number of operations per second that each DBMS can handle, while the x-axis displays the number of concurrent users. Each plot in the figure contains two series representing the two consistency levels used in the experiments: weak and strong.

Considering writing operations, we observe a significant impact on Cassandra's throughput, which drops by approximately 68% (from around 2600 to 800 requests) in a 10-user workload scenario when the data consistency level is switched from weak to strong. In the same scenario, MongoDB and Redis were less affected, with MongoDB experiencing a drop of about 36% (from around 1900 to 1200) at the worst case (with 30 users), and Redis showing an 8% decrease (from around 6200 to 5700) at its worst case (with 10 users). The significant performance drop in Cassandra's throughput can be attributed to its data sharding architecture, which distributes inserted data among nodes by token ranges. This adds complexity to operations as the coordinator node must wait for data allocation and replication before successfully responding. In contrast, MongoDB and Redis utilize a simpler primary-secondary architecture, where all primary data is stored on the primary node, which waits for secondary nodes to acknowledge new writes before confirming the operation. The results also indicate that Redis can handle more write operations per second than both Cassandra and MongoDB. However, Redis experiences a notable drop in throughput as the

Argument	Value	Description
headless	true	No interface execution.
spawn-rate	10	Users spawn rate.
users	Concurrent users variable	Number of concurrent users.
run-time	300 (seconds)	Duration of test execution.

TABLE 3 Locust arguments used.

[4] All the collected data can be consulted in: https://drive.google.com/file/d/1Xr43SsgmkhxZnJ6xj1cLGl4hy14r-w0J/view?usp=sharing

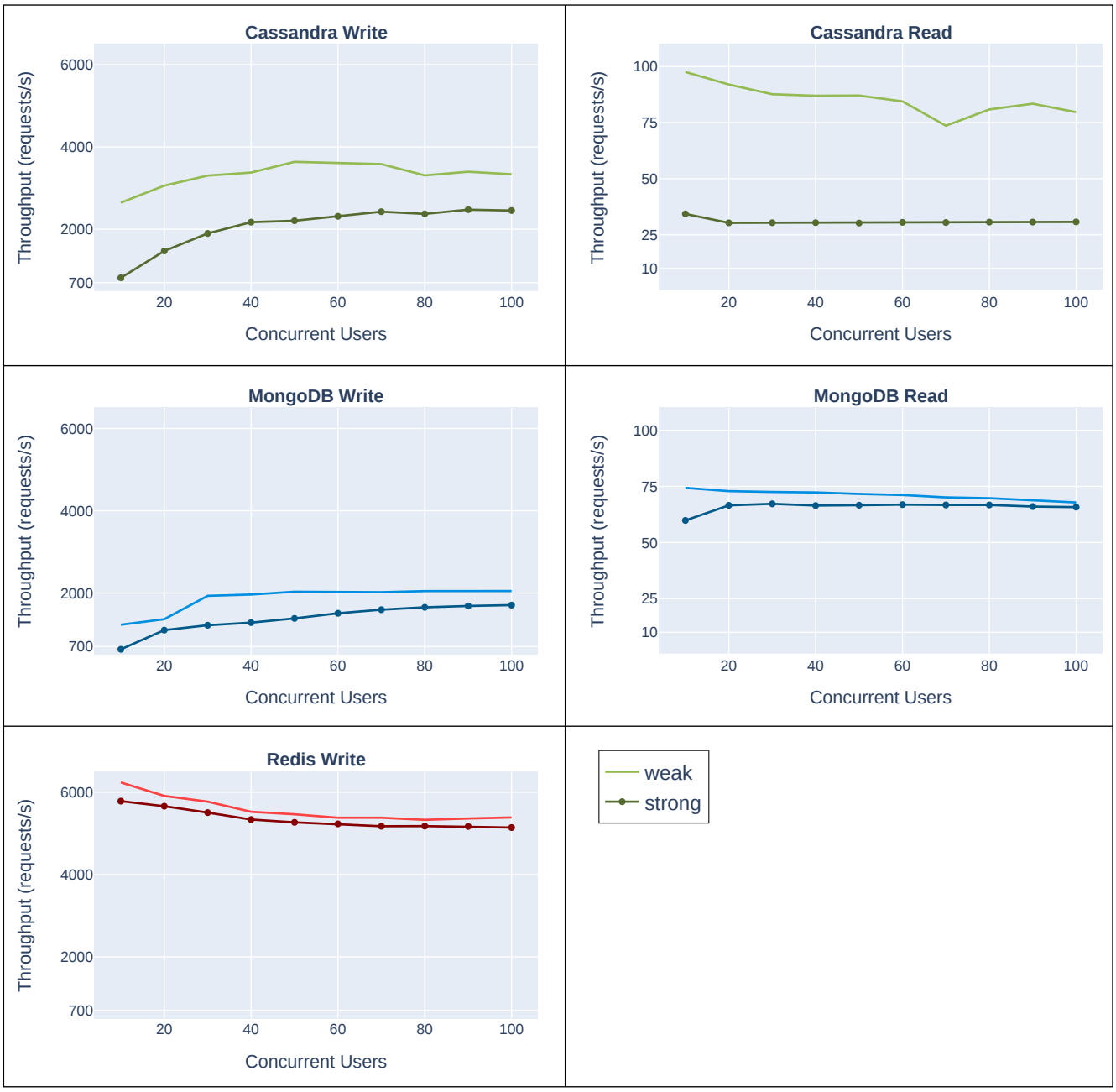

FIGURE 2 Comparison of throughput for all analyzed DBMSs across varying concurrent users and consistency levels (higher value on the y-axis is better).

number of concurrent users increases, whereas the other DBMSs remain relatively stable, handling nearly the same amount or even increasing the number of operations executed.

Considering reading operations, Cassandra's in-memory structure typically enhances reading performance under weak consistency compared to MongoDB. However, when adopting the strong data consistency level, Cassandra's performance drops by around 60% in all scenarios. Although MongoDB handles fewer requests per second than Cassandra when adopting the weak data consistency level, its performance is less affected when the data consistency level changes to strong, experiencing a

maximum throughput drop of 20% (from around 74 to around 59 requests per second) in a 10-users scenario. Both database management systems experienced a decrease in the number of reading operations handled per second as the number of concurrent users requesting data increased.

Throughput results also provide insights into the system's ability to handle the parallel execution of tasks. When a DBMS receives multiple requests simultaneously, its throughput may increase if it can effectively manage the workload. On the other hand, if the system struggles to handle concurrency, the number of requests per second may decrease. This behavior is evident in the results for Cassandra, where writing operations require less effort from the system, resulting in increased throughput with higher workloads, even when switching to the strong data consistency level. In contrast, the reading operations exhibit the opposite trend. The decrease in throughput observed in Figure 2 for Cassandra's reading operations suggests that the system may have been under stress due to concurrent requests and the additional validations required when adopting the strong data consistency level. We further discuss this issue by examining the results regarding response time variation across different data consistency levels in the next subsection.

5.2 | Response Time Analysis

Figure 3 illustrates the average response time (in milliseconds) for each database system under weak and strong consistency levels as a function of the number of concurrent users. The x-axis represents the number of threads, corresponding to the number of concurrent users, while the y-axis shows the average response time (where lower values indicate better performance).

Let us first analyze the response time variation related to consistency level changes (from weak to strong consistency). Redis exhibited the smallest difference in writing operations due to its in-memory structure, allowing quick write processing at runtime, with the largest difference seen at 30 concurrent users, where response time increased by around 16%, from 18 ms to 21 ms. In contrast, Cassandra showed the most significant variation, with writing operations increasing by over 240% (from 3.3 ms to 11.5 ms) with ten concurrent users, and reading operations rising by 224%, from around 480 ms to 1550 ms, with 50 concurrent users. MongoDB displayed a noticeable increase in response time for writing operations, especially with ten concurrent users, where the response time doubled from 7 ms (weak) to 15 ms (strong), while for reading operations, MongoDB experienced minimal variation, with increases under 10% across most scenarios.

Next, let us observe the overall impact of data consistency on response time. Strong consistency had the greatest impact across all scenarios, as expected, since the primary node must wait for confirmation from all secondary nodes before committing an operation, increasing response time. This effect can be more pronounced in geographically distributed setups, like cloud infrastructures, where network latency significantly affects performance. In the low-latency environment provided by our simulated local data center, the impact of strong consistency on response time was minimized, but once replicas are distributed across distant regions, the performance degradation would likely become more noticeable. Simultaneous increases in response time and drops in throughput were observed in all scenarios, highlighting a clear trade-off between consistency and response time. Overall, Redis and Cassandra performed better in writing operations, though Cassandra was most affected by switching consistency levels. MongoDB outperformed Cassandra in reading operations under strong consistency, given the same number of concurrent users across all DBMSs.

To assess the databases most affected by the consistency levels, we calculated the average percentage increase in response time for each database when transitioning from weak to strong data consistency across various workloads (see Figure 4). To determine the percentage increase for each workload scenario when changing from weak to strong data consistency, we used the formula $\frac{RT_{weak}}{RT_{strong}} - 1$, where RT_{strong} and RT_{weak} represent the measured response times under strong and weak data consistency levels, respectively. The results revealed that MongoDB experienced a response time increase of over 102% in writing operations with ten users, while Cassandra recorded an increase of over 200% under the same conditions. Cassandra was the most affected in all scenarios for both reading and writing operations, suggesting that MongoDB coordinates replication more efficiently, particularly in reading operations. Cassandra's data sharding mechanism impacts performance, primarily due to node latency, as the coordinator node must retrieve data from the node storing it before replication, adding overhead to the process.

5.3 | Statistical analysis of data consistency levels

This analysis focuses on whether the throughput and response time results differ significantly when using weak versus strong data consistency levels for each DBMS and operation. To assess this, we conducted hypothesis tests to determine if adopting a weak or strong consistency level had a statistically significant impact on database performance. We first evaluated whether the

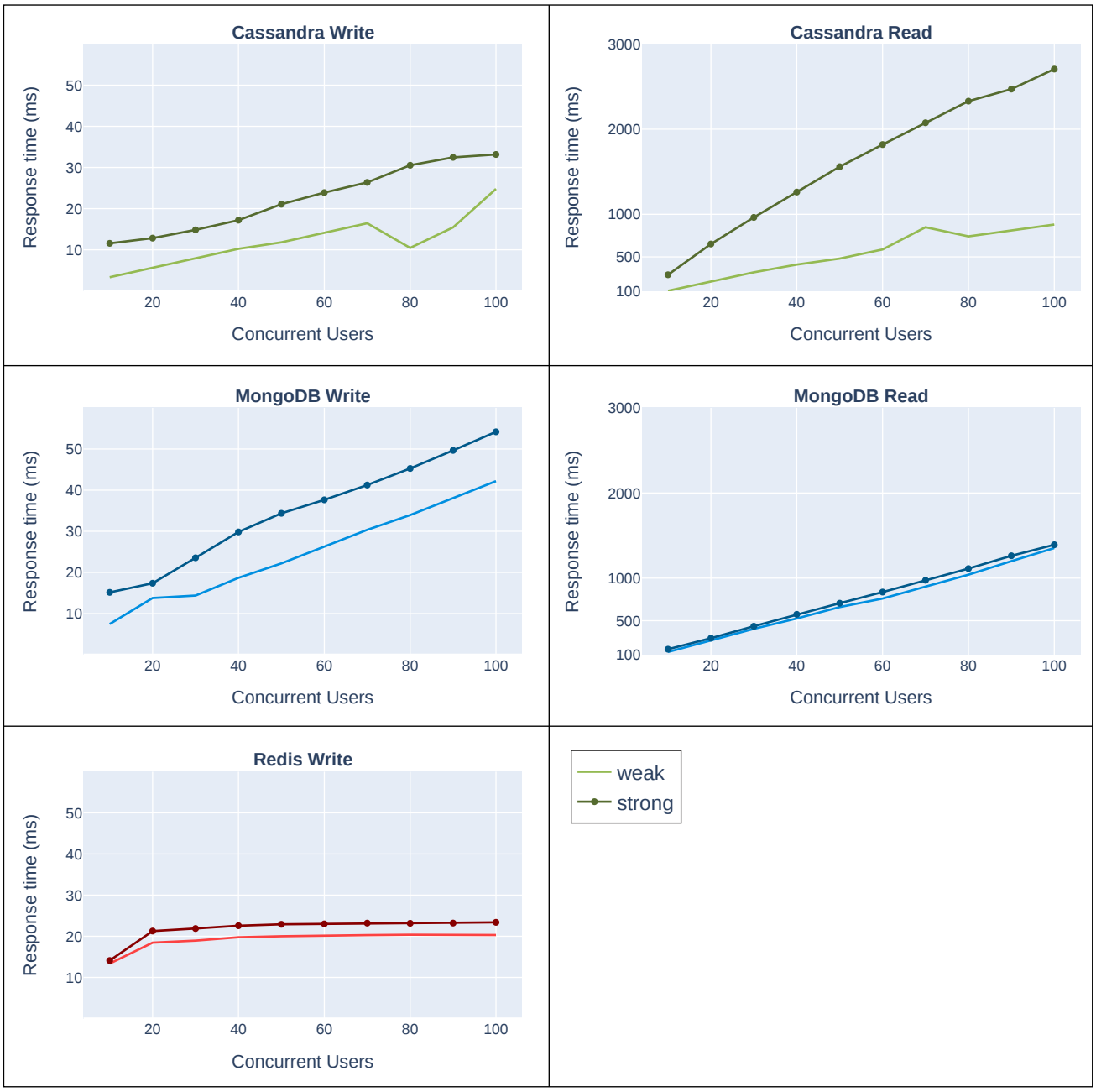

FIGURE 3 Comparison of response time for all analyzed DBMSs across varying concurrent users and consistency levels (lower value on the y-axis is better).

data followed a normal distribution using D'Agostino and Pearson's test[30]. For all cases, we considered a confidence interval of 95% (i.e., the p-value threshold was set to 0.05). We applied the Student's T-Test[31] for normally distributed data and the Mann-Whitney U Test[32] for non-normally distributed data.

The results are summarized in Table 4 . We highlight the p-value in bold when it is below 0.05 to indicate a statistical difference between adopting weak and strong data consistency levels. A statistical difference exists in 4 out of 5 scenarios analyzed regarding throughput results. Additionally, there is a statistical difference in 3 out of 5 scenarios when evaluating the average response time results. In summary, every combination of DBMS and operation demonstrates a statistically significant difference when switching from weak to strong consistency for at least one of the metrics analyzed.

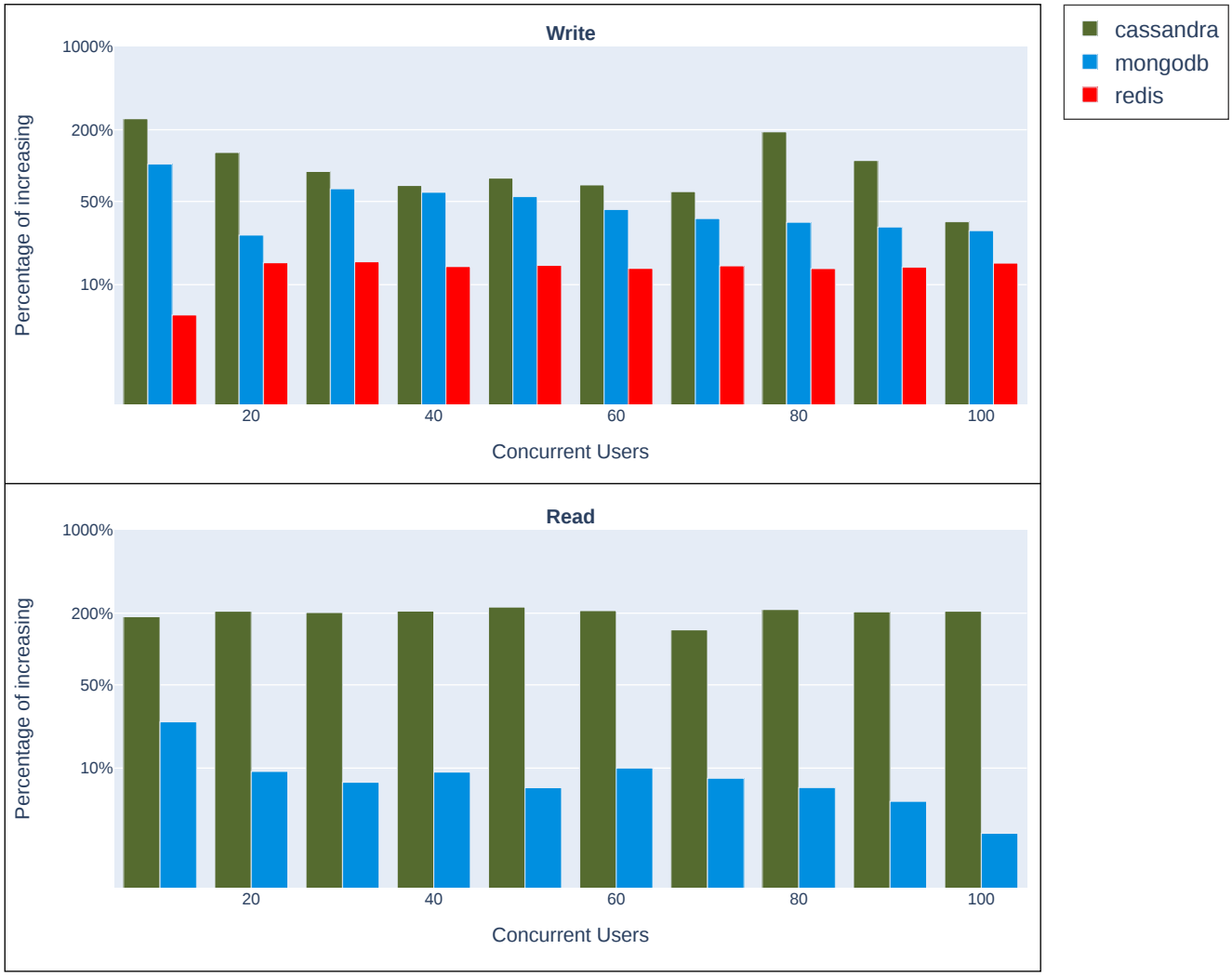

FIGURE 4 The percentage increase in response time when transitioning from weak to strong data consistency levels under different workloads for each database and operation.

Database	Operation	Throughput data normality	p-value	Response time data normality	p-value
Cassandra	Read	False	**0.000183**	True	**0.000918**
Cassandra	Write	False	**0.000183**	True	**0.004742**
MongoDB	Read	False	**0.000183**	True	0.785109
MongoDB	Write	False	**0.005795**	True	0.085583
Redis	Write	True	0.070539	False	**0.002202**

TABLE 4 Hypothesis test results.

5.4 | Threats to validity

In this Section, we discuss points that could influence the results of our experiments, and we shall focus on two main points: (*i*) the local infrastructure and network adopted and (*ii*) using a unique target host. We discuss these points as follows.

Local infrastructure and network. We adopted a local infrastructure and network to isolate our testbed from external factors such as network latency over the Internet and shared host (e.g., in the cloud). These are factors that could influence the results of the experiments. For instance, if cloud hosts are adopted, any instability in the connectivity of the source machine could affect the collected metrics. Therefore, having a controlled environment, such as the one adopted, helps us understand the actual performance of each DBMS utilized. Although we understand that real-world applications usually have geo-distributed replicas, using the local infrastructure, we could guarantee that all DBMSs analyzed had the same conditions for a fair comparison. In future work, we aim to analyze the impact of using a geo-distributed environment and compare the performance with our current experiments to understand how external factors could influence the results.

Unique target host. In our experiments, we utilized only one node to receive client requests, effectively designating it as the primary node throughout the tests. We recognize that Cassandra employs a masterless architecture, allowing any cluster node to handle incoming requests and perform the same functions as the others. In contrast, MongoDB and Redis operate on a primary-secondary model, where the primary node consistently acts as the request coordinator by default. Additionally, when configured as a cluster, Redis requires at least three master nodes, each with one replica. This cluster setup can enhance the concurrent execution of read and write operations. However, to standardize the experiments and ensure consistent conditions for a fair comparison among all analyzed DBMSs, as well as due to hardware availability limitations, we opted to configure only one node as the primary for all systems, and Redis was not configured as a cluster. We plan to investigate the impact of such configurations in future work.

6 | CONCLUSION AND FUTURE WORK

This work investigated the impact of data consistency levels on the performance of three popular NoSQL databases: Cassandra, MongoDB, and Redis. The results indicate that ensuring better consistency between nodes leads to a significant drop in performance, resulting in a decrease in the number of requests per second that can be handled. Our study highlights the trade-offs one should consider when selecting a database system and the desired data consistency level. We found that Redis performs exceptionally well in fast and consistent writing, while MongoDB performs better in intensive reading scenarios. In contrast, Cassandra outperforms both for weakly consistent writes, but its performance is significantly affected by stronger consistency levels. Furthermore, our experiments revealed statistically significant differences in throughput and response time for writing and reading DB operations when adopting different data consistency levels. These results provide valuable insights into the impact of consistency levels on the performance of these NoSQL databases and can assist in selecting the best database system for specific use cases. While similar studies often focused on single-database analyses, this research presented a detailed evaluation across multiple DBMSs, examining how each was impacted by changes in consistency levels and exploring key performance metrics across varying workloads.

In future research, we aim to conduct evaluations in geo-distributed or cloud-based database environments. We also plan to incorporate the YCSB benchmark[33] alongside Locust to enhance performance analysis. Additionally, we intend to explore experiments with a higher node count to provide a more comprehensive assessment, which could include configurations allowing for multiple nodes to handle client requests in systems like Cassandra.

References

1. Özsu MT, Valduriez P. Distributed and parallel database systems. *ACM Computing Surveys (CSUR)* 1996; 28(1): 125–128.

2. Nadiminti K, De Assunçao MD, Buyya R. Distributed systems and recent innovations: Challenges and benefits. *InfoNet Magazine* 2006; 16(3): 1–5.

3. Kemme B, Jiménez-Peris R, Patiño-Martínez M. Database replication. *Synthesis Lectures on Data Management* 2010; 5(1): 1–153.

4. Burckhardt S, others . Principles of eventual consistency. *Foundations and Trends® in Programming Languages* 2014; 1(1-2): 1–150.

5. Abadi D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 2012; 45(2): 37–42.

6. Bailis P, Venkataraman S, Franklin MJ, Hellerstein JM, Stoica I. Probabilistically bounded staleness for practical partial quorums. *arXiv preprint arXiv:1204.6082* 2012.

7. Gorbenko A, Romanovsky A, Tarasyuk O. Interplaying Cassandra NoSQL consistency and performance: A benchmarking approach. In: Springer. ; 2020: 168–184.

8. Diogo M, Cabral B, Bernardino J. Consistency models of NoSQL databases. *Future Internet* 2019; 11(2): 43.

9. Haughian G, Osman R, Knottenbelt WJ. Benchmarking replication in cassandra and mongodb nosql datastores. In: Springer. ; 2016: 152–166.

10. DB-Engines . DB-Engines Ranking. https://db-engines.com/en/ranking; 2024. Online; Visited 17-jan-2024.

11. Gomes C, Borba E, Tavares E, Junior MNdO. Performability Model for Assessing NoSQL DBMS Consistency. In: IEEE. ; 2019: 1–6.

12. Gomes C, O. Junior dMN, Nogueira B, Maciel P, Tavares E. NoSQL-based storage systems: influence of consistency on performance, availability and energy consumption. *The Journal of Supercomputing* 2023; 79(18): 21424–21448. doi: 10.1007/s11227-023-05488-6

13. Araújo C, Oliveira M, Nogueira B, Maciel P, Tavares E. Performability evaluation of NoSQL-based storage systems. *Journal of Systems and Software* 2024; 208: 111885. doi: 10.1016/j.jss.2023.111885

14. Tang E, Fan Y. Performance comparison between five NoSQL databases. In: IEEE. ; 2016: 105–109

15. Wang H, Li J, Zhang H, Zhou Y. Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra. In: Springer. ; 2014: 71–82.

16. Ferreira S, Andrade E, Mendonça J. Uma Abordagem Experimental para Avaliar os Níveis de Consistência do Banco de Dados NoSQL Cassandra. In: SBC. ; 2021: 156–167.

17. Brewer EA. Towards robust distributed systems. In: . 7. Portland, OR. ; 2000: 343–477.

18. Gilbert S, Lynch N. Perspectives on the CAP Theorem. *Computer* 2012; 45(2): 30–36.

19. Moniruzzaman A, Hossain SA. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191* 2013.

20. Strauch C, Sites ULS, Kriha W. NoSQL databases. *Lecture Notes, Stuttgart Media University* 2011; 20(24): 79.

21. Hewitt E. *Cassandra: the definitive guide*. " O'Reilly Media, Inc." . 2010.

22. Neeraj N. *Mastering Apache Cassandra*. Packt Publishing Ltd . 2015.

23. Membrey P, Plugge E, Hawkins T, Hawkins D. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Springer . 2010.

24. Abramova V, Bernardino J. NoSQL databases: MongoDB vs cassandra. In: ; 2013: 14–22.

25. Chen S, Tang X, Wang H, Zhao H, Guo M. Towards scalable and reliable in-memory storage system: A case study with Redis. In: IEEE. ; 2016: 1660–1667.

26. Heyman J, Byström C, Hamrén J, Heyman H. Locust.io. 2012.

27. Almesberger W, others . Linux network traffic control—implementation overview. 1999.

28. Ping C. Cloud Ping - AWS Latency Monitoring. https://www.cloudping.co/grid; 2024. Online; Visited 17-aug-2024.

29. Montgomery DC. *Design and analysis of experiments*. John wiley & sons . 2017.

30. DIAgostino R. An omnibus test of normality for moderate and large sample sizes. *Biometrika* 1971; 58(34): 1–348.

31. Kotz S, Johnson NL., eds.*The Probable Error of a Mean*: 33–57; New York, NY: Springer New York . 1992

32. McKnight PE, Najab J. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* 2010: 1–1.

33. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: ; 2010: 143–154.

34. Wiesmann M, Pedone F, Schiper A, Kemme B, Alonso G. Understanding replication in databases and distributed systems. In: IEEE. ; 2000: 464–474.

35. Bermbach D, Tai S. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In: ; 2011: 1–6.

36. Chandra DG. BASE analysis of NoSQL database. *Future Generation Computer Systems* 2015; 52: 13–21. doi: 10.1016/j.future.2015.05.003

DECLARATIONS

Availability of data and materials

The data that supports the findings of this study are available in the supplementary material of this article.

Competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

The authors declare that they did not receive any funding to produce this work.

How to cite this article: S. Ferreira, J. Mendonça, and E. Andrade (2016), Experimental Performance Analysis of Data Consistency Levels in NoSQL Databases.