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,

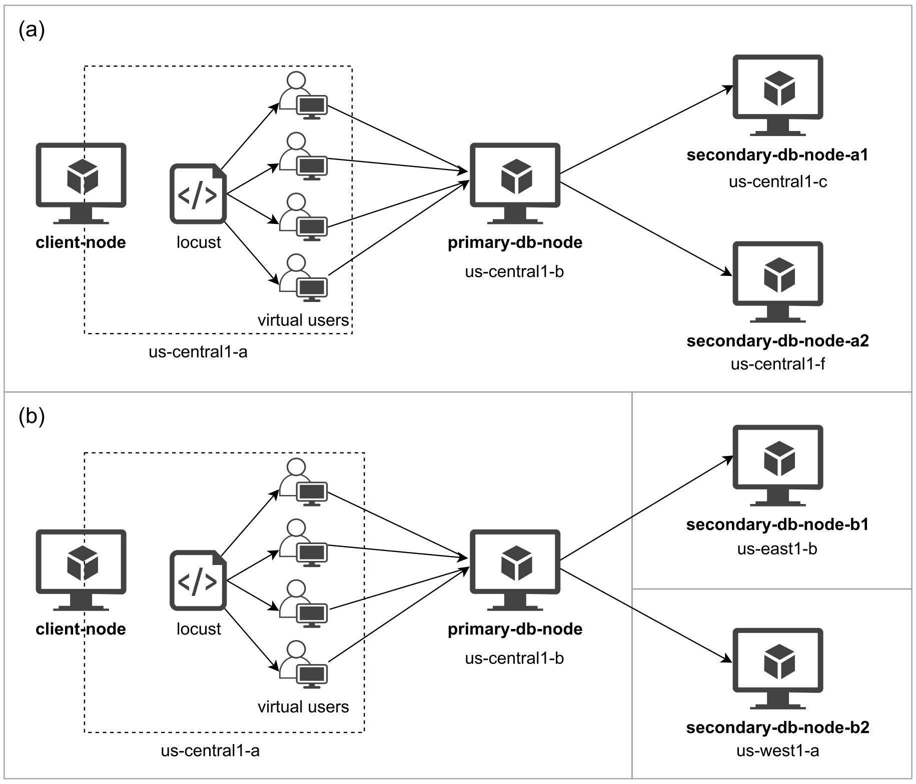

Fig. 1 Adopted testbed deployed on **a** a single-region (us-central1-a/b/c/f) and **a** multi-region (us-central1-a/b, us-east1-b, us-west1-a)

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.

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	

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.

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

[1] Cassandra Docker image link

[2] MongoDB Docker image link

[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

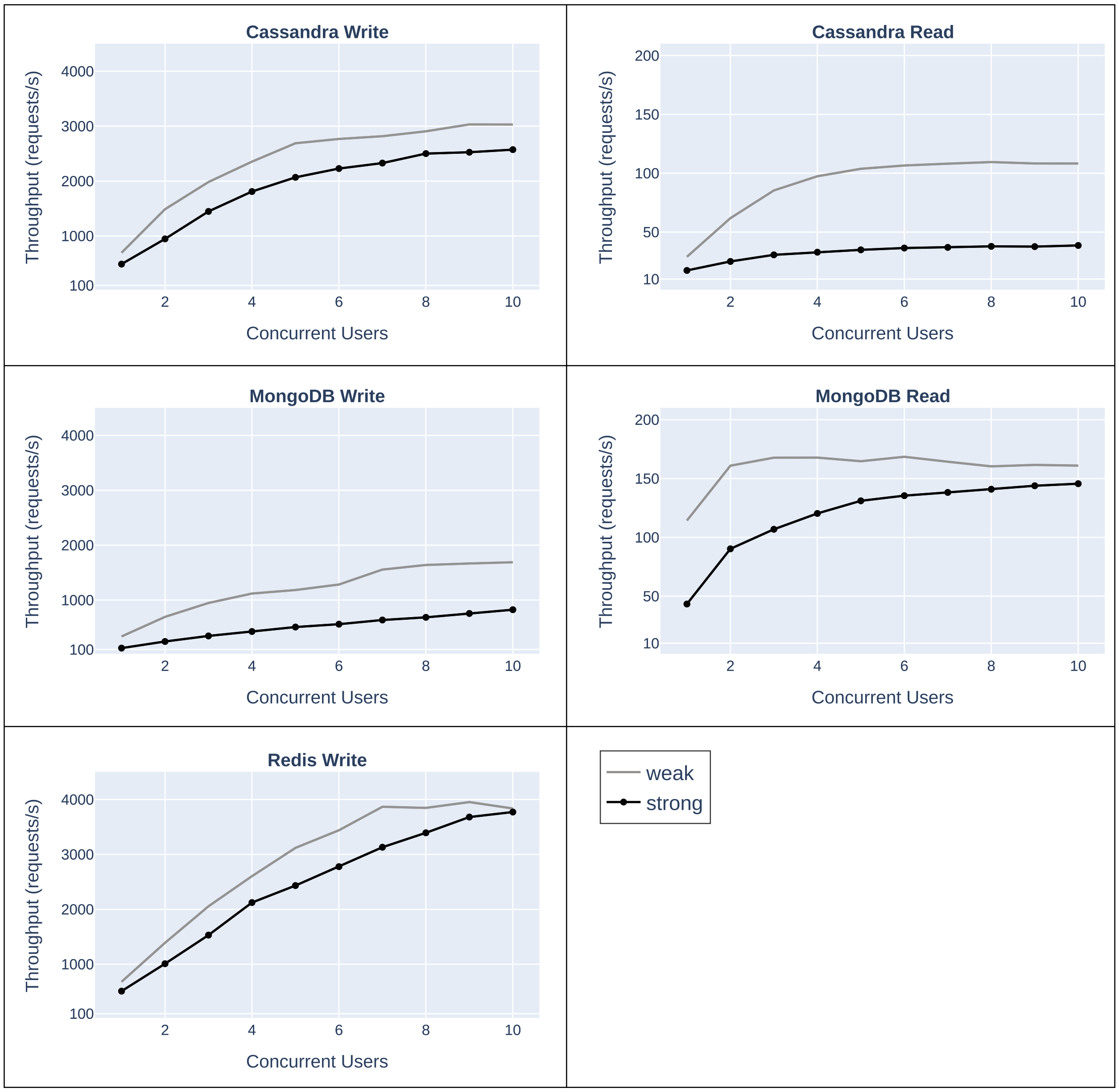

Fig. 2 Throughput results (in requests/s) for writing and reading operations of each NoSQL DBMS analyzed in single-region deployment. A *higher* value in the y-axis means *better* result. The plot scales differ on the y-axis for readability

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)

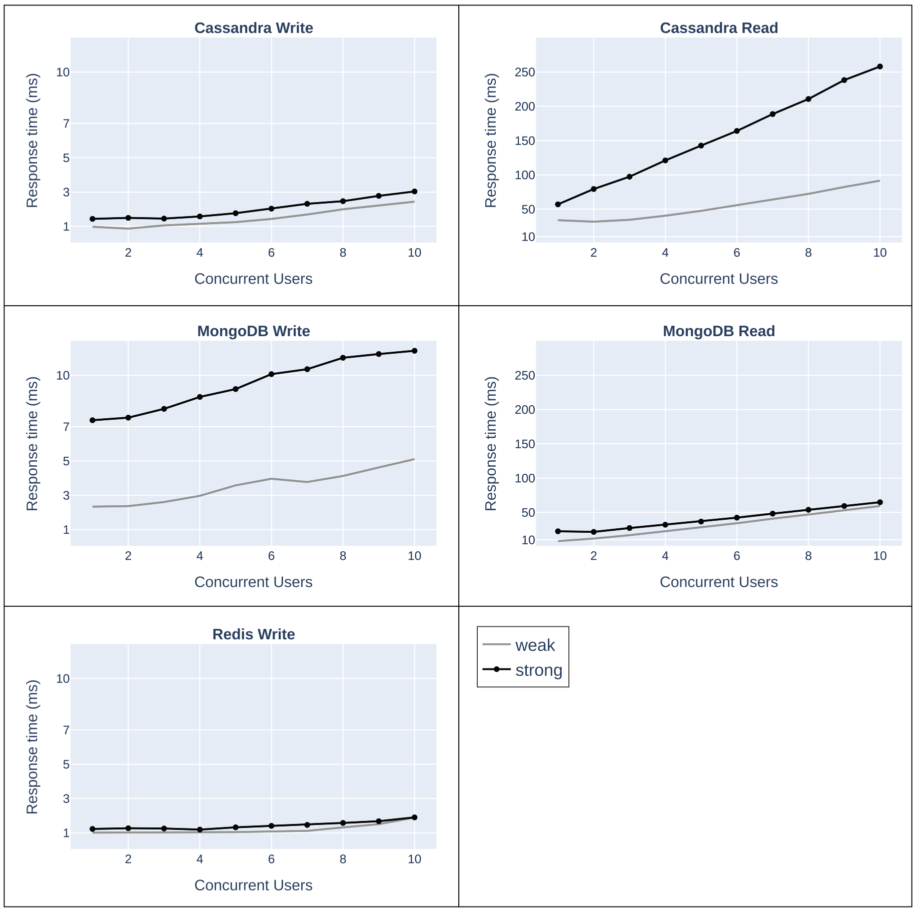

Fig. 3 Response time results for writing and reading operations of each NoSQL DBMS analyzed in single-region deployment. A *smaller* value in the y-axis means *better* result. Scales differ on the y-axis for readability

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

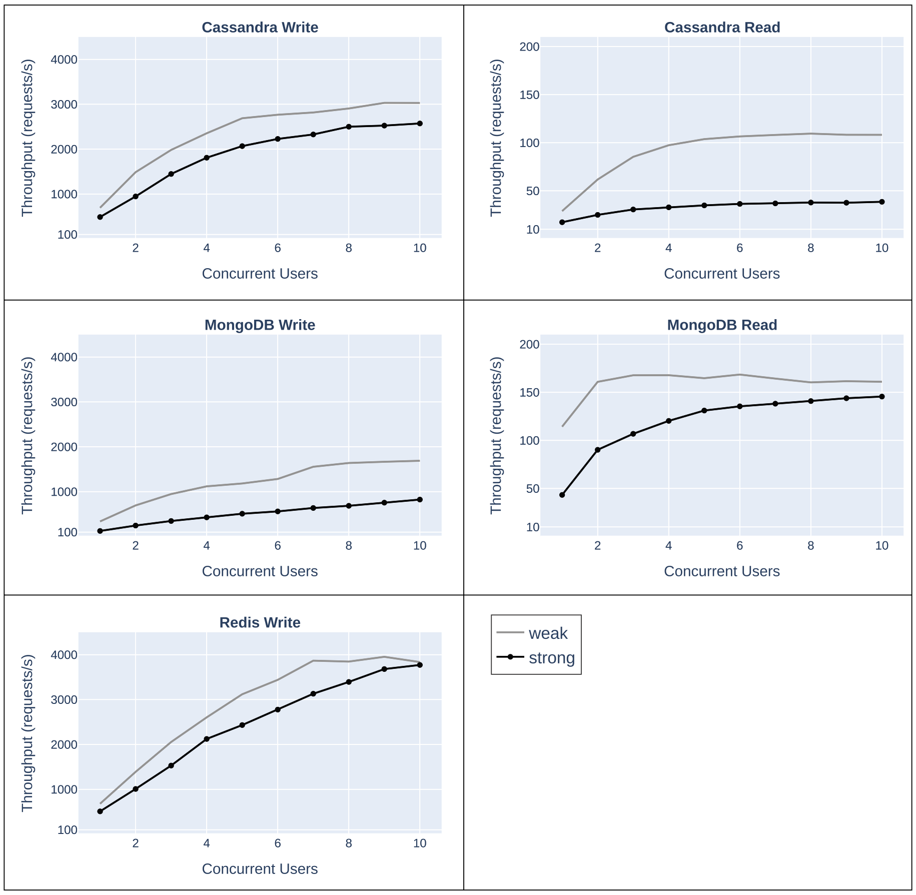

Fig. 4 Throughput results (in requests/s) for writing and reading operations of each NoSQL DBMS analyzed in multiple regions deployment. Scales differ on the y-axis for readability

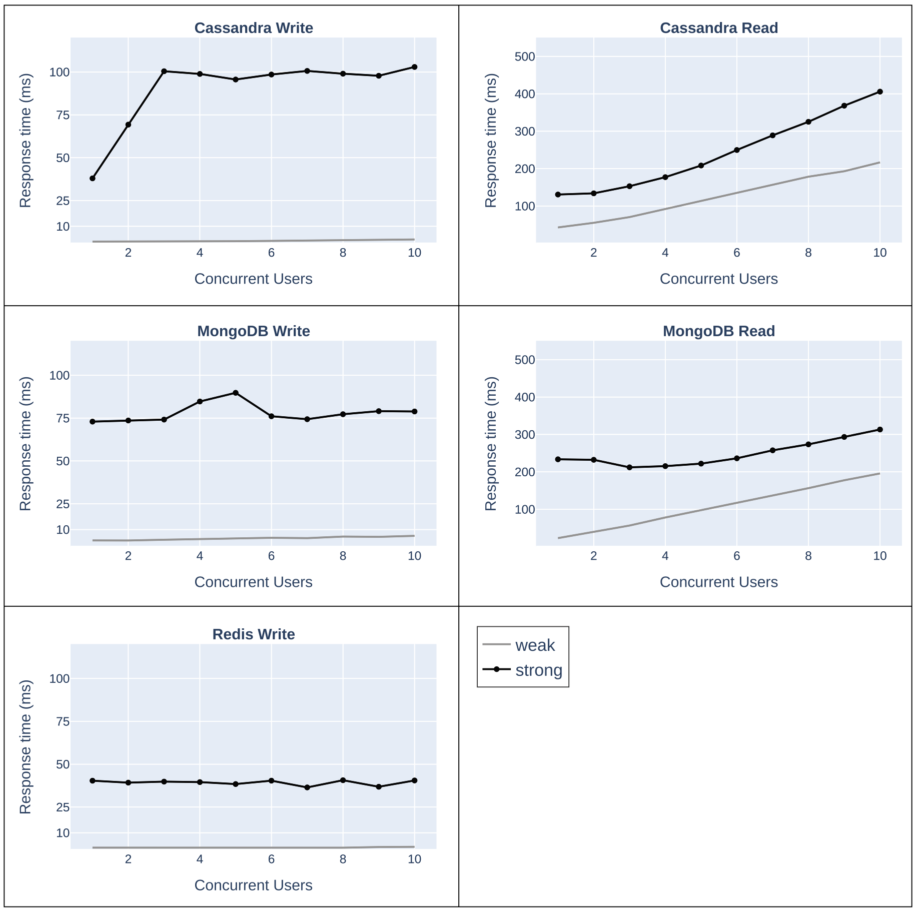

Fig. 5 Response time results (in ms) for writing and reading operations of each NoSQL DBMS analyzed in multiple regions deployment. Scales differ on the y-axis for readability

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.

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

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%

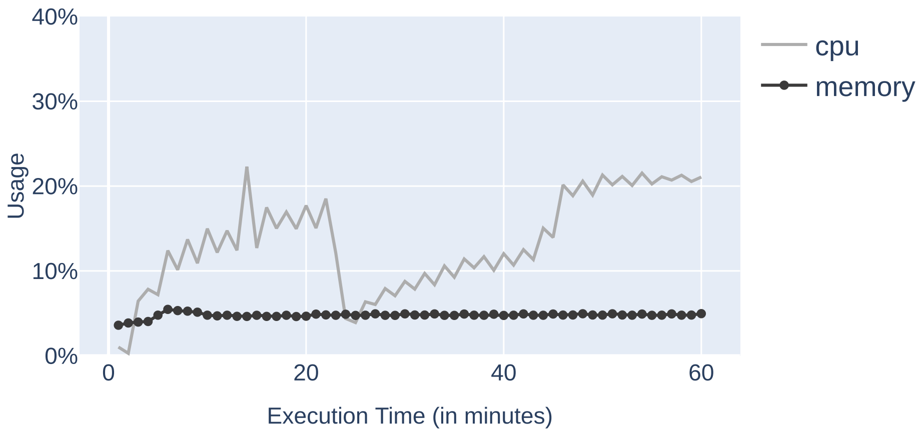

Fig. 6 client-node VM resource usage

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**

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.

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**

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. It shows that ensuring strong data consistency in multi-cloud regions could decrease the response time of a DBMS up to 4,360%. 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/154JvVfBWL6qeJ3K0Bs6eobPJPSjICzfc?usp=sharing.

Declarations

Competing interests
The authors declare no competing interests.

Received: 20 July 2024 Accepted: 20 October 2024
Published online: 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