International Journal Software Engineering and Computer Science (IJSECS)
5 (2), 2025, 691-705
Published Online August 2025 in IJSECS (http://www.journal.lembagakita.org/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v5i2.4206.

RESEARCH ARTICLE Open Access

Performance Analysis of NoSQL Databases: MongoDB Document Store and Redis Key-Value Store in Microservices-Based Applications Using Flask

Dava Ataya Shafi Andali *
Informatics Engineering Study Program, Faculty of Information Technology, Universitas Kristen Satya Wacana, Salatiga City, Central Java Province, Indonesia.
Corresponding Email: 672021186@student.uksw.edu.

Yeremia Alfa Susetyo
Informatics Engineering Study Program, Faculty of Information Technology, Universitas Kristen Satya Wacana, Salatiga City, Central Java Province, Indonesia.
Email: yeremia.alfa@uksw.edu.

Received: May 7, 2025; Accepted: July 10, 2025; Published: August 1, 2025.

Abstract: The research examines performance characteristics between two NoSQL database architectures: MongoDB as a document-oriented system and Redis as an in-memory key-value store, implemented within microservices applications developed using Flask framework. Growing enterprise requirements for scalable, high-performance systems drive increased adoption of NoSQL databases paired with microservices architectures. The investigation assesses database performance through systematic CRUD and aggregation operations executed on nested data structures that mirror real-world public transportation datasets. Redis demonstrates superior operational efficiency in real-time scenarios, attributed to its memory-resident architecture. Empirical findings reveal Redis maintains consistently reduced response latencies compared to MongoDB across virtually all tested operations. Read operations show Redis achieving 0.00037-second average execution times, representing a 60.22% performance improvement over MongoDB's 0.00093-second baseline. Read-by-ID queries exhibit more pronounced differences, with Redis completing operations in 0.00105 seconds against MongoDB's 0.00873 seconds—an 87.96% performance differential. Update and delete operations demonstrate Redis execution times of 0.00026 and 0.00028 seconds respectively, compared to MongoDB's 0.00088 and 0.00087 seconds, yielding approximately 70% and 68% performance advantages. Delete-all operations reveal substantial disparities: Redis completes bulk deletions in 0.083 seconds while MongoDB requires 0.27 seconds, representing a 69.26% performance penalty. Aggregation functions including summation, minimum, and maximum value calculations follow similar performance patterns, with Redis executing operations more efficiently across all test scenarios. Performance evaluations were conducted on Windows 10 Pro 64-bit (Build 19045) equipped with 15.8 GB memory and an 11th Generation Intel® Core™ i5-11400H processor featuring 6 cores and 12 threads. Testing utilized MongoDB version 1.45.4 and Redis version 2.66, with hardware specifications directly influencing benchmark outcomes. Results indicate Redis optimization for applications demanding high-performance real-time data access, while MongoDB serves applications requiring flexible document storage capabilities and complex data structure management.

Keywords: NoSQL; MongoDB; Redis; Microservices; Flask; Performance Benchmarking; Document Database; Key-value Store; Real-time Data Processing; CRUD Operations.

1. Introduction

NoSQL database adoption has surged rapidly due to modern application demands requiring more flexible and scalable data storage solutions. NoSQL databases like Redis and HBase are widely utilized in applications requiring high performance and the ability to handle large workloads, as demonstrated in research on NoSQL database benchmarking using Yahoo Cloud Serving Benchmark [1]. MongoDB, one of the leading NoSQL systems, attracts many developers because of its capability to handle unstructured data through a highly flexible document-based data model. Such flexibility facilitates data adaptation across various application scenarios like e-commerce and real-time analytics, which demand high flexibility in data management [2]. Alongside software architecture evolution, many large companies such as Netflix and Amazon have shifted from monolithic architectures to microservices. Microservices architecture offers better horizontal scalability and flexibility in managing independently distributed applications. Nevertheless, transitioning to microservices also brings new challenges, including greater resource consumption and increased complexity in inter-service data management. The transition has become a primary choice in the technology industry because it provides performance improvements and resilience, as outlined in performance evaluations between monolithic and microservices architectures [3]. Considering these developments, the general trend in modern application development is adopting NoSQL technologies and microservices architectures to meet high-performance requirements, scalability, and flexibility in data management. However, challenges related to resource consumption and integration complexity also become important factors that need consideration.

Problems arising from NoSQL and microservices technology development indicate urgent challenges regarding performance, flexibility, and architecture management. In NoSQL selection, choosing the right system becomes crucial because each NoSQL type offers different specific advantages. Imbalance between performance and application needs can hinder efficiency and scalability, especially in big data-based applications [1]. MongoDB, popular with its document-based data model, provides flexibility that facilitates application development. However, such flexibility can also become a weakness in applications requiring high data consistency [2]. Meanwhile, microservices provide significant advantages such as faster development cycles and better scalability, but on the other hand also pose architecture degradation risks. Decentralized systems with module boundaries that are difficult to understand can cause performance degradation and misalignment with initial design [4]. These challenges demonstrate the importance of proper technology selection and the need for thorough analysis of architectural changes, so risks can be minimized and optimal performance of NoSQL and microservices-based systems can be maintained.

The technology solution offered in the analysis involves using MongoDB and Redis to address flexible and scalable microservices-based application needs. MongoDB, as a document database, provides high flexibility in managing unstructured data, while Redis, with its in-memory architecture, offers superior performance in real-time operations. Both technologies are combined to maximize performance, where MongoDB handles complex data and Redis processes high-speed data [5][6]. The Flask framework is used to integrate both databases within microservices architecture, enabling lightweight and separate service development. The main problem faced is scale and performance, where MongoDB excels in data flexibility, while Redis is faster in real-time operations. The benchmarking will provide guidance for developers in selecting appropriate technology, based on specific microservices-based application needs [4][5].

NoSQL technology and microservices architecture development creates challenges related to performance, flexibility, and system management. MongoDB, with high flexibility in handling unstructured data, often faces data consistency issues, while Redis excels in high-speed real-time operations. In microservices applications, using a combination of MongoDB and Redis offers solutions that optimize performance and scalability. Flask serves as the framework for integrating both databases, ensuring separate services can be managed efficiently. Performance benchmarking of both will help developers choose the right technology according to specific application requirements.

2. Related Work

Comparative studies on NoSQL database management systems have established foundational benchmarks for performance evaluation across different database architectures. The experimental work by researchers examining Performance Comparison Between Multi-Model, Key-Value And Documental Nosql Database Management Systems, evaluated query response times across ArangoDB, OrientDB, Couchbase, Redis, and MongoDB. Their findings revealed substantial performance variations, with ArangoDB demonstrating inferior performance metrics compared to alternative NoSQL implementations. The research specifically addressed enterprise decision-makers and developers seeking empirical data on average query response times for database selection processes [7]. Building upon multi-model database evaluations, the current investigation narrows its scope to document-oriented MongoDB and key-value Redis databases. While

previous research examined broader NoSQL ecosystem comparisons, including multi-model architectures like ArangoDB alongside traditional NoSQL variants, the present study concentrates on two distinct NoSQL paradigms that represent dominant approaches in microservices implementations.

REST API framework performance analysis has become increasingly relevant as microservices adoption accelerates across enterprise environments. Purwanto's comparative analysis of Flask, Laravel, and Express.js frameworks revealed significant performance differentials in API development scenarios. The study documented Flask's response time range of 723-1757 milliseconds, which exceeded competitor frameworks' performance metrics. However, Flask demonstrated zero error rates during data request processing, establishing reliability as a distinguishing characteristic. These findings underscore the trade-off between response speed and system stability in framework selection for high-availability applications [8]. The empirical evidence from Purwanto's research establishes Flask as a viable foundation for microservices architecture despite response time limitations. The framework's error-free operation under load testing conditions provides confidence for implementing Flask-based microservices that integrate NoSQL databases. The stability metrics become particularly relevant when evaluating database integration performance, as framework reliability directly impacts overall system performance measurements.

Database performance evaluation in distributed systems has received considerable attention, particularly regarding Redis and MongoDB implementations. Febriyani, Pramukantoro, and Bakhtiar conducted systematic performance analysis comparing Redis, Mosquitto, and MongoDB as message brokers within IoT middleware architectures. Their methodology examined reliability, scalability, and resource utilization efficiency across varying data volumes and operation types [9]. The experimental results demonstrated Redis superiority in CPU utilization during write operations, memory efficiency, and runtime performance, establishing its position as an optimal solution for high-throughput data ingestion scenarios. Conversely, MongoDB exhibited superior disk I/O performance and read operation efficiency, particularly under large-scale data retrieval workloads. These performance characteristics directly inform database selection strategies for microservices applications requiring specific read/write operation profiles.

Existing literature establishes performance baselines for individual technologies but lacks focused analysis of MongoDB and Redis integration within Flask-based microservices architectures. Previous studies either examine broader NoSQL comparisons across multiple database types or evaluate framework performance independently of database integration scenarios. The current research addresses this gap by specifically examining MongoDB and Redis performance within Flask microservices implementations. The positioning of this investigation builds upon established performance metrics from prior research while extending analysis to practical microservices deployment scenarios. By leveraging Flask's demonstrated stability characteristics and applying them to NoSQL database integration testing, the study aims to provide empirical guidance for technology stack selection in modern distributed application development.

3. Research Method

This research employs a methodology to analyze NoSQL database performance, specifically MongoDB and Redis, in Flask-based microservices applications using two main approaches: a Flowchart that illustrates the overall research workflow and an Incremental Model used in system development.

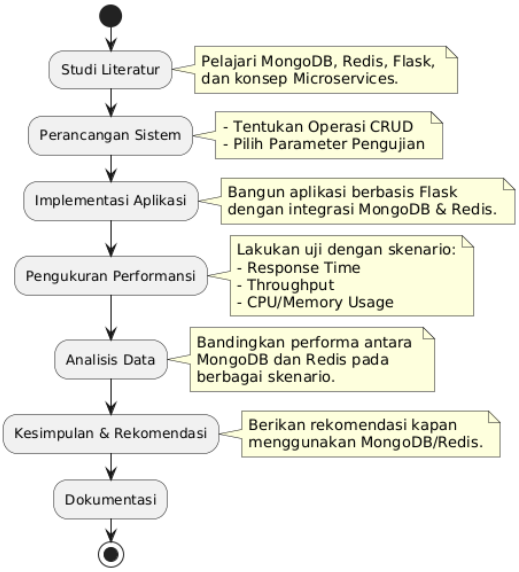

Figure 1. Flowchart

The research flowchart depicts the steps taken in the process of analyzing and comparing performance between MongoDB and Redis. Beginning with literature study, the first step involves learning MongoDB, Redis, Flask, and microservices concepts to understand each technology that will be used. Following that, in the system design phase, CRUD operations to be tested are selected along with determining relevant testing parameters, such as response time, throughput, and CPU or memory usage [10]. Additionally, the research considers technical aspects that affect testing results by including detailed testing system specifications: Windows 10 Pro 64-bit operating system (Build 19045), 15.8 GB RAM, 11th Gen Intel(R) Core™ i5-11400H processor with 6 cores and 12 threads. The database versions used are MongoDB 1.45.4 and Redis 2.66. The next step is application implementation, where Flask-based applications are built by integrating MongoDB and Redis as data storage systems. After the application is built, performance measurement is conducted by testing various scenarios, including response time, throughput, and resource usage. The amount of data tested reaches thousands of entries arranged in nested structures, resembling real public transportation data. For response time measurement, a millisecond-based timestamp approach is used with the time.time() function on the application side to record start and end times of each operation, obtaining accurate execution duration for each scenario. Data obtained from testing is then analyzed in the data analysis phase, where MongoDB and Redis performance are compared across various test scenarios conducted. Based on the analysis results, in the conclusion and recommendation stage, recommendations are formulated regarding when to use MongoDB or Redis in microservices-based applications. Finally, research results are documented to provide a clear picture of findings and recommendations generated during the research.

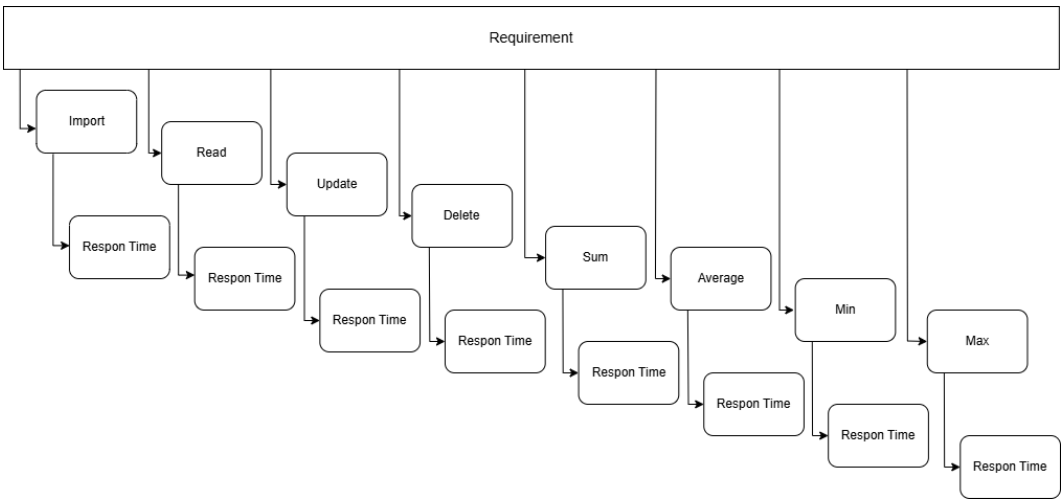

Figure 2. Incremental Model

Besides the flowchart, this research also adopts an incremental model in system development to test MongoDB and Redis performance gradually. The incremental model provides advantages because it allows gradual system development with evaluation conducted at each phase. The authors chose to use the incremental model because they expected feedback at each incremental stage to determine response time for each operation [11]. The incremental model shown in the diagram above illustrates a gradual approach in system development to meet previously defined requirements. In the initial stage, system requirements are determined as the foundation of the entire development process. The model divides into several stages or increments, where each stage aims to systematically add specific features or functions to the system.

The first stage begins with import, which is the process of entering data into the system. After that, testing is conducted on the system's ability to read data through the read process, followed by the ability to update data through update, and delete data with delete. At each stage, response time is always measured to evaluate system performance in handling operations performed. After basic operations (import/create, read, update, delete), the next stage focuses on calculation operations, including summation (sum), average (average), minimum value (min), and maximum value (max), always measuring response time for each calculation operation as a performance indicator. This approach has advantages in ensuring that each added feature has been thoroughly tested before proceeding to the next stage. Furthermore, response time measurement at each increment allows for evaluating system efficiency and identifying potential performance issues at each development stage. However, in using the incremental model, the authors focus on measuring response speed for each operation to compare response times between MongoDB and Redis databases in handling performed operations.

4. Result and Discussion

4.1 Results

The testing conducted to determine the comparison of response time speed from create/insert, read, read by, update, delete, delete all commands and aggregation operations consisting of sum, min, max on Flask-based microservice applications is presented in the form of time calculation tables for each command and graphs.

4.1.1 System Architecture

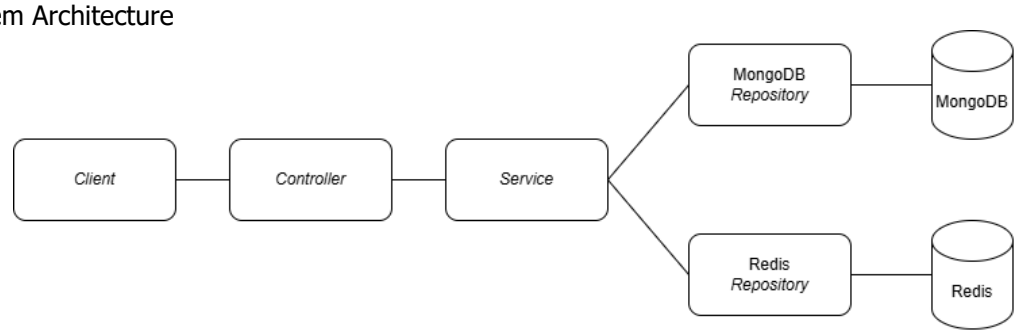

Figure 3. System Architecture

The architecture diagram shows the system workflow in a microservices-based application that uses Flask as the backend and integrates two types of NoSQL databases: MongoDB and Redis. The workflow starts from the Client sending requests to the Controller, which receives, validates, and forwards requests to the Service for further processing. The Service handles business logic and determines data storage destinations, whether to MongoDB or Redis, according to system requirements. To interact with each database, the Service utilizes MongoDB Repository and Redis Repository, which function as abstraction layers for database operations. MongoDB Repository manages interactions with MongoDB as a document-based database, while Redis Repository handles operations with Redis, which is a key-value database. With this architectural structure, the system can manage data more efficiently, enabling clear separation of responsibilities between each part, and improving scalability and flexibility in handling various data types in microservices-based applications.

4.1.2 Data Structure

The data used in system testing consists of public transportation information including stations, routes, and bus or train departure schedules. The data is structured in nested format, allowing hierarchical information storage with interconnected attributes within a single entity. In this research, MongoDB and Redis are compared regarding storage efficiency and data retrieval speed on such structures. MongoDB, as a document-based database, provides flexibility in storing nested data within one document without normalization requirements, while Redis, as a key-value store database, stores data in key-value pairs, which is more optimal for fast data retrieval. Testing was performed by measuring both databases' performance in handling storage and data search operations with predetermined scenarios. Through this comparison, we expect to gain understanding about the advantages and disadvantages of each database in handling complex structured data in microservices-based applications.

The data used in this research has a nested or hierarchical structure, which is a form of data representation where one entity can have one or more sub-entities arranged hierarchically. This structure is commonly used in NoSQL databases like MongoDB and Redis to store and manage data with complex relationships more efficiently. Nested structure allows more organized data storage without requiring normalization like relational databases. For example, in this research, the data consists of main entities that have several attributes which can also be objects or lists of other objects. This facilitates data access and manipulation as a single unit without needing joins between tables like in RDBMS (Relational Database Management System). In MongoDB, data is stored in JSON document format, which naturally supports nested structures, allowing storage of nested parts within one document without breaking them into separate tables. Meanwhile, in Redis, data is typically stored in key-value format, and to handle nested structures, data can be serialized in JSON format before being stored as strings.

4.1.3 Code Implementation

Code Program 1. Controller for receiving and processing JSON Files

```
@data_bp.route('/insert', methods=['POST'])
def upload():
  if 'file' not in request.files:
      return jsonify({"error": "No file part in the request"}), 400
```

```
  file = request.files['file']

  if file.filename == '':
      return jsonify({"error": "No file selected for uploading"}), 400

  if file and file.filename.endswith('.json'):
      try:
          data = json.load(file)

          mongo_result = mongo_service.insert(data)
          redis_result = redis_service.insert(data)

          return jsonify({
              "mongo_result": mongo_result,
              "redis_result": redis_result
          }), 200
      except json.JSONDecodeError as e:
          return jsonify({"error": f"Invalid JSON format: {str(e)}"}), 400
  else:
      return jsonify({"error": "Only JSON files are allowed"}), 400
```

The main parts of the code 'mongo_service.insert(data)' and 'redis_service.insert(data)' represent parallel data insertion processes into two types of NoSQL databases: MongoDB and Redis. This represents an implementation of polyglot persistence strategy, which uses more than one storage technology to leverage each one's strengths. Redis is used because of its ability to process data quickly thanks to in-memory architecture, very suitable for real-time needs. On the other hand, MongoDB excels in storing data with complex structures thanks to its document schema flexibility. The advantage of this approach is the combination of speed and flexibility, ideal for microservices-based systems. However, this approach also brings risks of data inconsistency if one of the insert processes fails, so additional strategies are needed to ensure data synchronization between databases [12].

Code Program 2. Service for storing data to MongoDB

```
def insert(self, data):
  if not isinstance(data, list):
      raise ValueError("Data must be a list")

  start_time = time.perf_counter()

  try:
      inserted_count = self.repository.insert_many(data)
      end_time = time.perf_counter()
      duration = end_time - start_time

      return {
          "message": "Data successfully stored to MongoDB",
          "inserted_count": inserted_count,
          "upload_time_seconds": duration
      }

  except Exception as e:
      return {
          "message": str(e),
          "inserted_count": 0,
          "upload time seconds": None
      }, 500
```

Code Program 3. Repository for storing data to MongoDB

```
def insert_many(self, data):
  result = self.collection.insert_many(data)
  return len(result.inserted_ids)
```

In code program 3, the insert(self, data) function aims to insert large amounts of data into MongoDB using the 'insert_many' method. This function measures data insertion execution time, providing information about the number of successfully inserted data and the time required. This approach utilizes batch insertion which is more efficient than one-by-one insertion, which can increase throughput and reduce connection overhead [13].

Code Program 4. Service for storing data to Redis

```
def insert(self, data):
  start_time = time.perf_counter()
  try:
      inserted_count = self.repository.insert_many(data)
      end_time = time.perf_counter()
      duration = end_time - start_time
      return {
          "message": "Data successfully stored to Redis",
```

```
            "inserted_count": inserted_count,
            "upload_time_seconds": duration
        }

    except Exception as e:
        return {"message": str(e), "inserted_count": 0, "upload_time_seconds": None}, 500
```

Code program 4 has the insert(self, data) function that inserts data in batches to Redis using the 'insert_many' method and measures its execution duration. This approach leverages Redis's advantages as an in-memory database that offers high speed in read and write operations. With the time measurements performed, we can evaluate how fast Redis can process large amounts of data, which is very relevant for applications requiring real-time performance. Using batch insertion also increases efficiency compared to one-by-one insertion operations, reducing overhead and speeding up throughput [13].

Code Program 5. Repository for storing data to Redis

```
def insert_many(self, data):
    pipeline = self.client.pipeline()
    inserted_count = 0

    if isinstance(data, list):
        for item in data:
            key = item['_id']
            value = msgpack.packb(item)
            pipeline.set(key, value)
            inserted_count += 1
    else:
        key = data['_id']
        value = msgpack.packb(data)
        pipeline.set(key, value)
        inserted_count = 1

    pipeline.execute()
    return inserted_count
```

The 'insert_many(self, data)' function utilizes Redis pipeline to perform batch data insertion. Pipeline allows atomic execution of multiple Redis commands, reducing communication overhead between application and Redis server. Each inserted data is processed using 'msgpack.packb' to convert data into a more efficient format before storage. This is very useful when working with large amounts of data, as it increases throughput and reduces latency. By using pipeline, this function ensures that many Redis operations are executed in one batch, optimizing execution time compared to performing operations one by one [14].

4.1.4 System Specifications

Table 1. System Testing Specifications

Component	Details
Database Type	MongoDB and Redis
Database Kind	NoSQL Document MongoDB, Key Value Redis
MongoDB Version	1.45.4
Redis Version	2.66
Operating System	Windows 10 Pro 64-bit (Build 19045)
System Memory	15.8 GB
CPU Type	Intel® Core™ i5-11400H (11th Gen)
Total Cores	6
Total Threads	12

Table 1 presents the system specifications used as the environment for conducting all experiments.

4.1.5 Database Operation Test Results

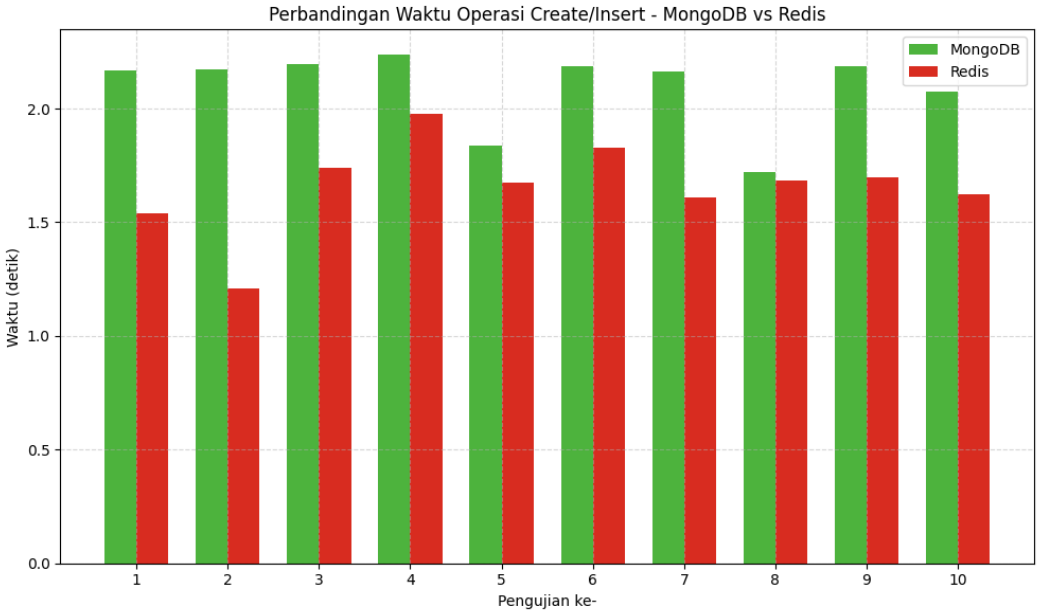

Figure 4. Create Data Results

Based on the test results shown in Figure 4, Redis consistently shows faster execution time compared to MongoDB in 10 trials of create/insert operations. Redis, as an in-memory database, utilizes data storage in RAM resulting in low latency, with execution times ranging from 1.2 to 2 seconds. Meanwhile, MongoDB, which is disk-based, requires longer time, between 1.7 to 2.2 seconds. This difference aligns with the basic characteristics of both databases, where Redis is more optimal for real-time applications with fast access needs, while MongoDB excels in data storage flexibility in document format [15].

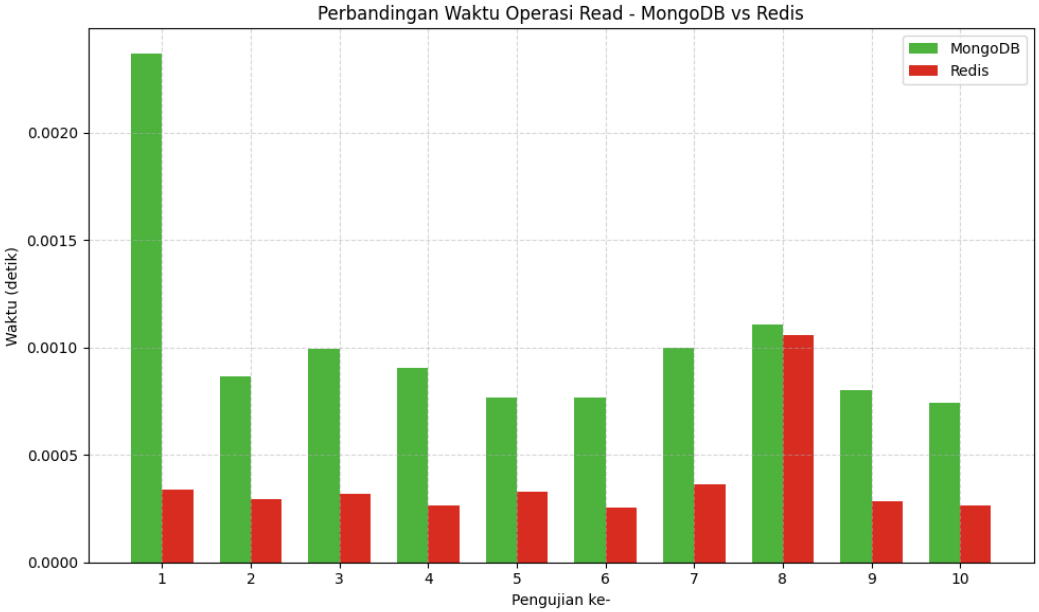

Figure 5. Read Data Results

Based on Figure 5, which shows the comparison of read operation times between MongoDB and Redis in 10 tests, it appears that Redis consistently has lower read operation times compared to MongoDB. This indicates that Redis has better performance in read operations compared to MongoDB. This performance difference can be explained by Redis's in-memory architecture, which allows faster data access compared to MongoDB which stores data on disk [5].

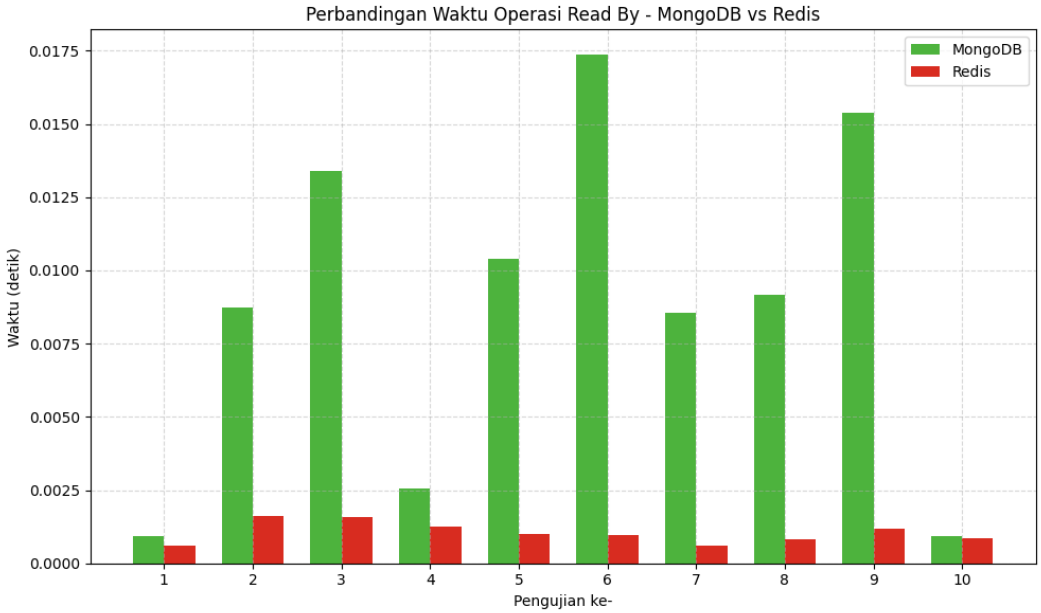

Figure 6. Read by Data Results

Figure 6 shows the comparison of read by operation execution times between MongoDB and Redis based on stoptimes data for one travel route. Test results show that Redis is consistently faster than MongoDB in every trial. Redis, as an in-memory database with key-value structure, can access data very quickly. Conversely, MongoDB requires longer time because it uses BSON (Binary JSON) format to store documents, which although flexible, requires parsing processes and matching more complex data structures when data searches are performed [16].

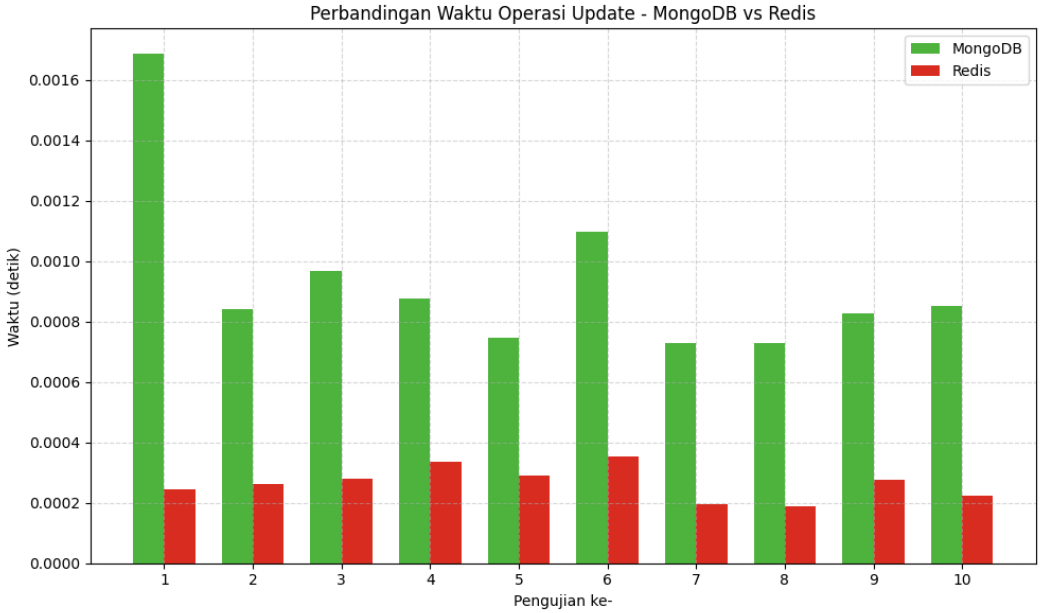

Figure 7. Update Data Results

Based on test results in the graph, Redis shows faster and more consistent update operation response times compared to MongoDB in almost all tests. This indicates that Redis is more efficient in handling update requests in the tested scenarios, and has higher stability levels. This performance shows that Redis is more suitable for applications requiring real-time data access speed compared to MongoDB [17].

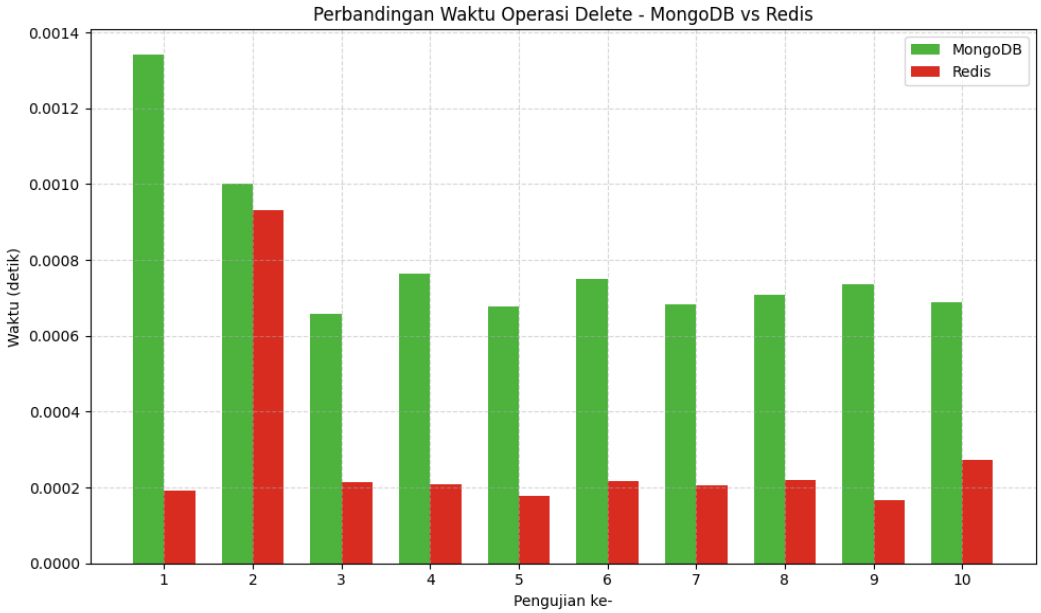

Figure 8. Delete Data Results

Delete operations are one of the key functions in data management that affects database performance efficiency. Testing shows that Redis has faster execution time compared to MongoDB in performing data deletion based on ID. This is caused by the simple key-value structure used by Redis, which allows deletion to be performed directly without additional search processes. Meanwhile, document-based MongoDB requires a search process first to find the appropriate document before deletion is performed. Although MongoDB supports indexes, its more complex data structure and consistency mechanisms maintained during deletion cause higher execution time compared to Redis. Generally, architectural differences between both systems become the main factor in delete operation performance differences, where Redis is superior for fast and high-scale data deletion needs [18].

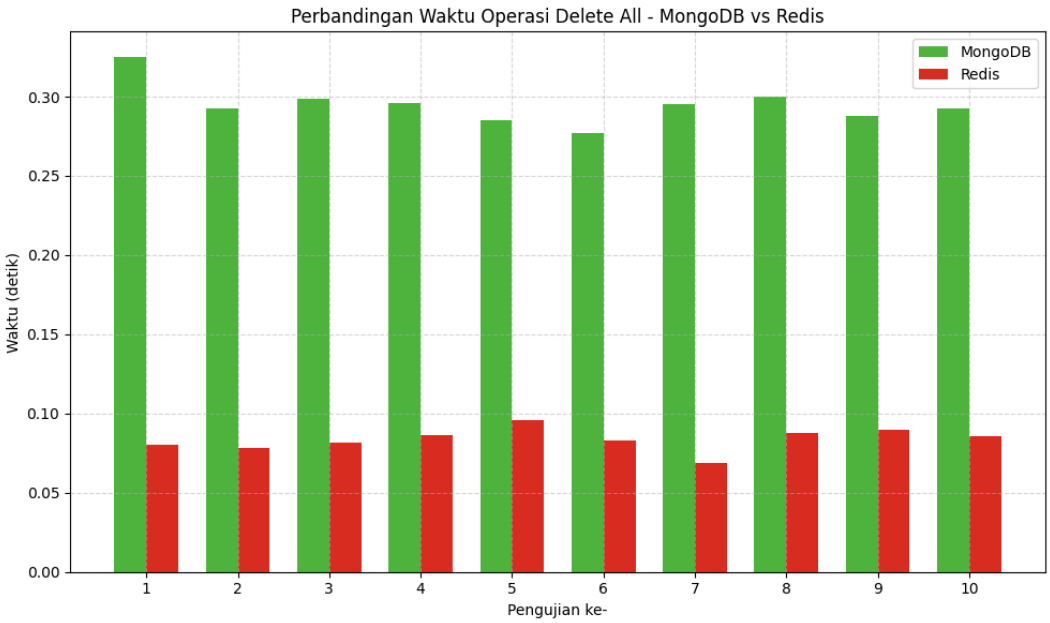

Figure 9. Delete All Data Results

Figure 9 shows that Redis has much faster execution time for deleting all data compared to MongoDB. This is caused by Redis's memory-based architecture, enabling instant data deletion through FLUSHALL command without requiring additional processing. Conversely, MongoDB requires longer time because data deletion is performed with commands like deleteMany({}), which still must traverse documents and update internal structures like indexes. This process adds execution load, so the required time becomes higher compared to Redis which is architecturally simpler [19].

4.1.6 Aggregation Operation Test Results

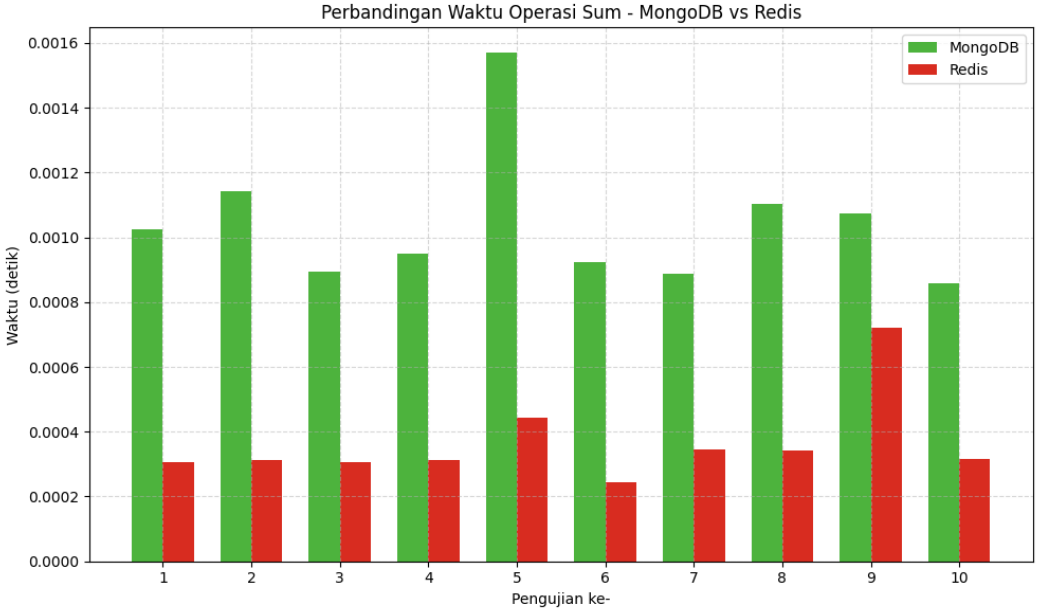

Figure 10. Sum Data Results

Figure 10 shows the comparison of sum operation execution times (total sum of routes data) between MongoDB and Redis in 10 tests. This sum operation calculates total values from all entries in a column, which is used to measure data aggregation performance. Results show that Redis (red bars) consistently has faster execution time compared to MongoDB (green bars), with average time below 0.0004 seconds, while MongoDB is in the range of 0.0009 to 0.0016 seconds. This difference is caused by Redis's in-memory based architecture, which makes it more efficient in processing numerical data aggregately compared to MongoDB which uses disk-based storage. These findings show that Redis is superior in terms of speed for simple aggregation operations like sum, and becomes a more appropriate choice for applications requiring high performance in real-time data processing [20].

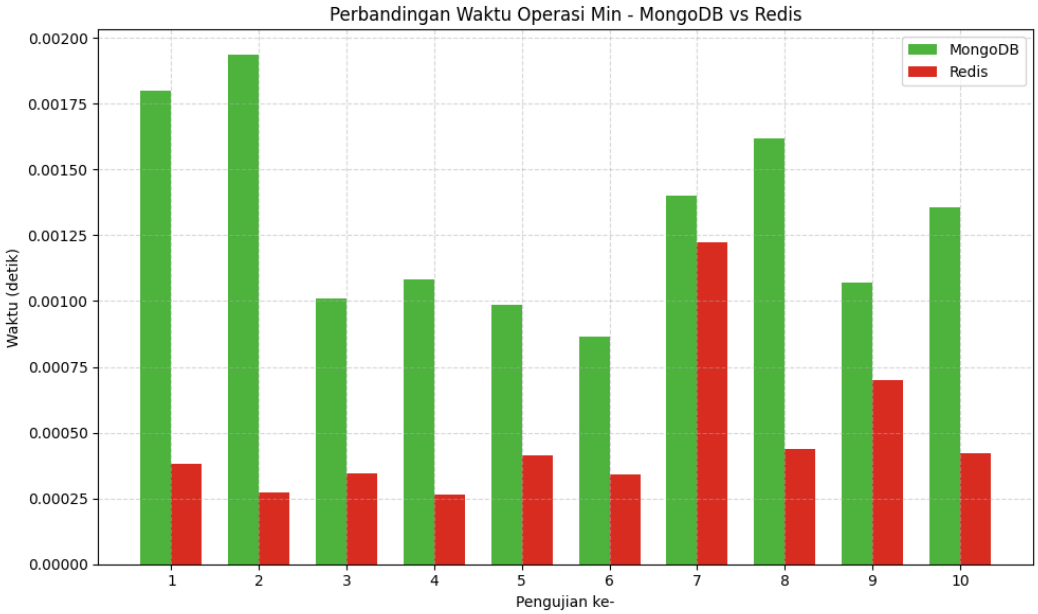

Figure 11. Min Data Results

Figure 11 shows the comparison of min operation times between MongoDB and Redis. Min operations aim to retrieve the lowest value from a data set, commonly used in statistical analysis or performance tracking. Redis (red bars) shows faster execution time compared to MongoDB (green bars), with quite significant differences especially in several early tests. This performance difference occurs because Redis stores all data in memory so it can access and compare values instantly, while MongoDB must read and process data from disk, causing higher latency. This shows that Redis is more efficient in comparative operations like min, especially when application scenarios require fast and repeated value searches [21].

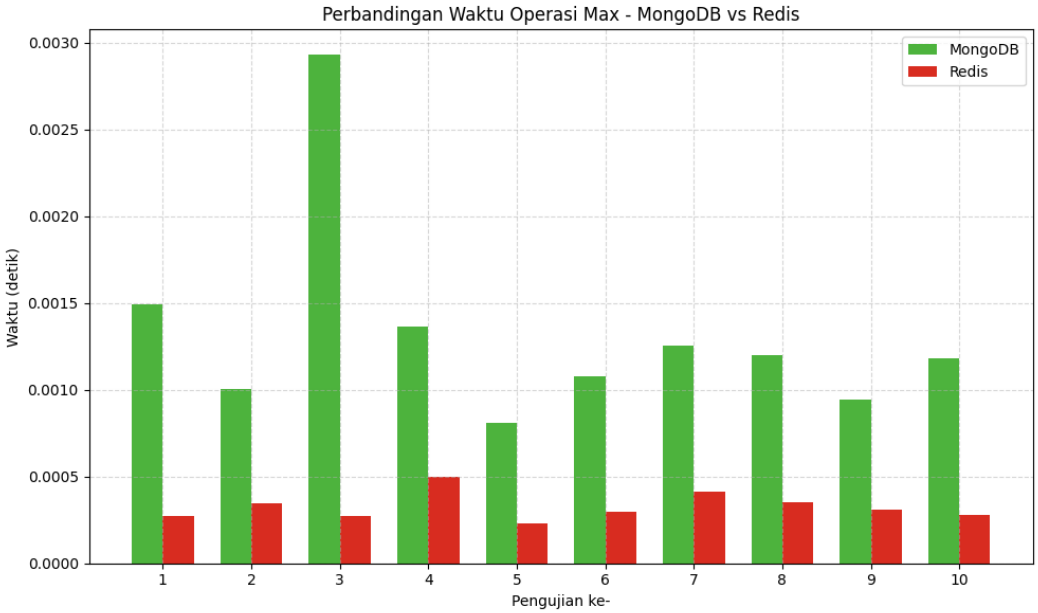

Figure 12. Max Data Results

The figure above shows the comparison of max operation execution times (searching for the largest value from routes data) between MongoDB and Redis in 10 tests. Max operations function to determine the highest value from a data set, and are commonly used in data analysis such as recording maximum performance or upper value limits. Based on the graph, Redis (red bars) consistently shows lower execution time compared to MongoDB (green bars). This difference is influenced by how each system manages and processes data: Redis uses simple and efficient data structures like sorted sets that allow extreme value searches to be performed quickly, while MongoDB must perform traversal and aggregation on documents with more complex structures. Additionally, Redis also relies on lightweight and direct command execution through pipeline, while MongoDB involves more complex query stages. This makes Redis more responsive for operations like max that require fast comparative processing [22].

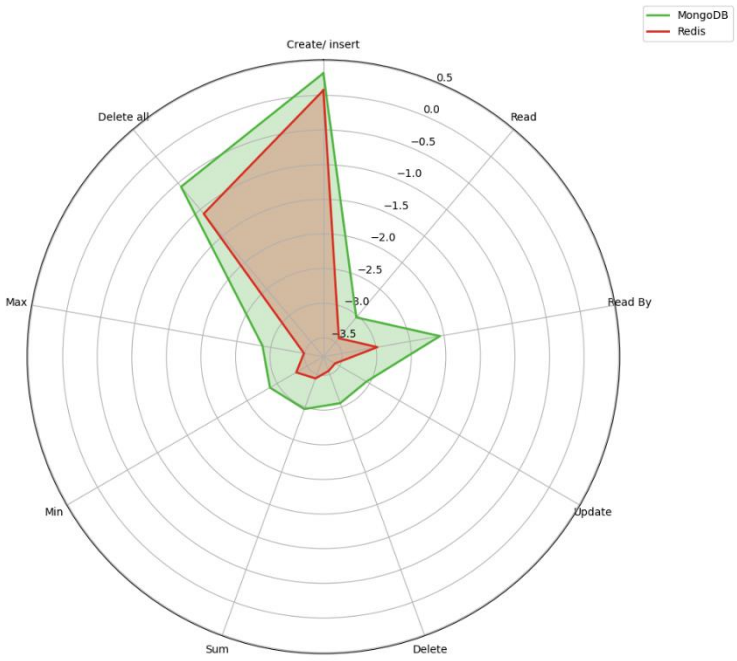

Figure 13. Average Data

The radar chart above shows the comparison of average execution times for various types of operations between MongoDB and Redis. Operations tested include create/insert, read, read by ID, update, delete, delete all, and aggregation operations like sum, min, and max. From test results, it appears that each point on the chart represents the average execution time of each operation in seconds. Values on the graph have also been

logarithmically transformed so that differences between values are easier to visualize, considering there are quite significant gaps between certain operations like insert and delete. In the create/insert section, MongoDB recorded an average time of around 2.09 seconds, while Redis was at around 1.18 seconds. For read and read by ID operations, Redis shows smaller values compared to MongoDB, respectively around 0.00037 seconds and 0.00105 seconds, compared to MongoDB which requires around 0.00093 seconds and 0.00873 seconds. Similar patterns also appear in update and delete operations, where Redis recorded lower times. Redis recorded times of around 0.00026 seconds for update and 0.00028 seconds for delete, while MongoDB recorded 0.00088 seconds and 0.00087 seconds. In aggregation operations like sum, min, and max, Redis also shows smaller execution times, with quite significant time differences compared to MongoDB. Finally, in delete all operations, Redis recorded time around 0.083 seconds, while MongoDB required around 0.27 seconds.

4.2 Discussion

Based on the code implementation above, data storage is performed through an API endpoint that receives JSON files and stores them to both MongoDB and Redis to analyze their performance in microservices architecture. The implementation follows the Controller-Service-Repository pattern, which separates responsibilities in the code program for better structure. The Controller receives JSON files, validates their format, and forwards them to the Service Layer. When the file is valid, data gets sent to both MongoDB and Redis for storage. In MongoDB, data storage utilizes the insert_many() method, allowing multiple documents to be stored in one operation. Execution time is recorded using time.perf_counter(), and storage results return the number of successfully stored documents along with process duration. Redis employs a key-value approach where each entry maps based on _id as the key, while data is stored in MessagePack format as the value. Storage uses pipeline functionality, enabling multiple operations to run simultaneously for increased efficiency and reduced latency. After storage completion, the system returns the number of successfully stored entries with their execution time. This approach expects Redis to achieve higher storage speed through pipeline optimization, while MongoDB offers more flexibility in handling complex data structures. The implementation serves as the foundation for analyzing performance comparisons between MongoDB and Redis in microservices architecture.

The test results demonstrate that Redis consistently outperforms MongoDB across almost all CRUD operations. Several technical factors explain this performance difference. Redis utilizes in-memory architecture that enables data access with extremely low latency, whereas MongoDB relies on disk-based storage requiring additional I/O operations. The simple key-value structure in Redis allows direct data access without complex parsing processes, contrasting with MongoDB's BSON format that necessitates query parsing procedures. Redis pipeline implementation enables batch operation execution, reducing network communication overhead significantly. These architectural differences create substantial performance gaps between the two database systems, particularly in scenarios requiring rapid data access and manipulation.

For aggregation operations including sum, min, and max functions, Redis maintains its performance advantage over MongoDB. In-memory processing allows mathematical operations to execute directly without disk I/O requirements, resulting in faster computation times. Redis provides specialized data structures like sorted sets that are specifically optimized for comparative and aggregation operations. The single-threaded event loop architecture in Redis reduces context switching overhead during command processing, contributing to improved performance metrics. MongoDB's document-based structure requires more complex traversal and aggregation processes, leading to longer execution times for similar operations. The performance gap becomes particularly evident in scenarios requiring frequent aggregation calculations or real-time data analysis.

While Redis demonstrates superior performance across various operations, several trade-offs and implementation considerations must be evaluated. Redis faces memory limitations as it's constrained by system RAM capacity, whereas MongoDB can handle significantly larger datasets through disk-based storage. MongoDB provides better data durability with automatic persistence mechanisms, while Redis requires specific configuration for data persistence functionality. MongoDB excels in handling complex queries and relational operations due to its document-based structure and query language capabilities. The dual-database implementation strategy introduces challenges in maintaining data consistency between systems, requiring additional synchronization mechanisms. Organizations must weigh these factors against performance benefits when choosing between these database technologies. The decision should align with specific application requirements, including data volume, query complexity, performance needs, and persistence requirements. Redis proves ideal for applications prioritizing speed and real-time performance, while MongoDB suits scenarios requiring complex data relationships and larger storage capacity.

5. Conclusion and Recommendations

Research comparing two NoSQL database types - MongoDB (document-based) and Redis (key-value based) - reveals that Redis consistently delivers faster execution times than MongoDB across nearly all tested operations within Flask-based microservices applications. Testing covered various operations including create/insert, read, read by ID, update, delete, delete all, plus aggregation functions (sum, min, max), with results visualized through radar charts using logarithmic scales to demonstrate significant performance differences. Redis recorded an average of 1.18 seconds for insert operations, outpacing MongoDB's 2.09 seconds. For read and read by ID operations, Redis achieved 0.00037 seconds and 0.00105 seconds respectively, substantially faster than MongoDB's 0.00093 seconds and 0.00873 seconds. Update and delete operations also favored Redis at 0.00026 seconds and 0.00028 seconds, compared to MongoDB's 0.00088 seconds and 0.00087 seconds. Even for delete all operations, Redis excelled with 0.083 seconds versus MongoDB's 0.27 seconds. Redis also demonstrated superior efficiency in aggregation operations like sum, min, and max calculations. Redis achieves superior performance primarily through its in-memory architecture, making it ideal for real-time systems requiring low latency and high throughput. MongoDB remains valuable for systems needing flexible data structures and semi-structured schema support, though at the cost of higher execution times. Database selection in microservices architecture should align with primary application needs - whether prioritizing data access speed or schema flexibility. Redis works best for session storage, caching, and real-time message queues, while MongoDB suits applications requiring flexible data structures like content management systems, e-commerce platforms, or complex document storage.

Research limitations include the absence of horizontal scalability, failover, and security testing, meaning results may not reflect performance in large-scale production environments. Future work could expand testing to include memory usage and throughput analysis, plus implementation in large-scale scenarios with complex microservices configurations. High-concurrency load testing and integration with technologies like Kafka or other message brokers could provide additional research directions. Security analysis and cross-service transaction consistency also warrant consideration for developing optimal and sustainable system architecture recommendations.

References

[1] Alzaidi, M., & Vagner, A. (2022). Benchmarking Redis and HBase NoSQL databases using Yahoo Cloud Service Benchmarking tool. *Annales Mathematicae et Informaticae*, *56*, 1–9. https://doi.org/10.33039/ami.2022.12.006

[2] Rathore, M., & Bagui, S. S. (2024, September). MongoDB: Meeting the dynamic needs of modern applications. *Encyclopedia*, *4*(4), 1433–1453. https://doi.org/10.3390/encyclopedia4040093

[3] Blinowski, G., Ojdowska, A., & Przybylek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, *10*, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

[4] Bushong, V., Abdelfattah, A. S., Cerny, T., Taibi, D., Lenarduzzi, V., & Khomh, F. (2021, September 1). On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, *11*(17), Article 7856. https://doi.org/10.3390/app11177856

[5] Kausar, M. A., Nasar, M., & Soosaimanickam, A. (2022, August). A study of performance and comparison of NoSQL databases: MongoDB, Cassandra, and Redis using YCSB. *Indian Journal of Science and Technology*, *15*(31), 1532–1540. https://doi.org/10.17485/IJST/v15i31.1352

[6] Syach, U., & Edi, S. W. M. (2024). Perancangan Aplikasi Web Manajemen Data Produk Bisnis Perhiasan Berbasis Flask Dan Mongodb. *IT-Explore: Jurnal Penerapan Teknologi Informasi dan Komunikasi*, *3*(2), 162-176. https://doi.org/10.24246/itexplore.v3i2.2024.pp162-176.

[7] Jansson, J., & Vukosavljevic, A. (2021). *Performance comparison between multi-model, key-value and documental NoSQL database management systems* [Bachelor's thesis, University of Skövde].

[8] Purwanto, T. (2023). Analisa Perbandingan Kinerja Rest Api Dengan Framework Flask, Laravel, Dan Express Js. *Scientia Sacra: Jurnal Sains, Teknologi dan Masyarakat*, *3*(4), 49-55.

[9] Febriyani, F., Pramukantoro, E. S., & Bachtiar, F. A. (2019). Perbandingan Kinerja Redis, Mosquitto, dan MongoDB sebagai Message Broker pada IoT Middleware. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, *3*(7), 6816-6823.

[10] Mutmainnah, A., Musyrifah, M., & Zulkarnaim, N. (2022). Perbandingan Relational Database dan Non-Relational Database dalam Pengembangan Smart Tourism. *Jurnal Teknik Informatika dan Sistem Informasi*, *8*(1), 150-160. https://doi.org/10.28932/jutisi.v8i1.4353

[11] Andriani, E. O., Wahyuni, E. D., Ramadha, F. N., & Alfarizy, A. Z. (2024). Analisa Perbandingan Software Development Model Antara Metode Scrum Dan Metode Incremental. *Jurnal Media Informatika dan Teknologi*, *2*(1).

[12] Kazanavičius, J., Mažeika, D., & Kalibatienė, D. (2022, June). An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture: A case study for mainframe database. *Applied Sciences*, *12*(12), Article 6189. https://doi.org/10.3390/app12126189

[13] Thapa, A. B. (2022). *Optimizing MongoDB performance with indexing–Practices of indexing in MongoDB* [Bachelor's thesis, Degree programme in Information and Communications Technology]. https://urn.fi/URN:NBN:fi:amk-2022061317593.

[14] Tallberg, S. (2020). *A comparison of data ingestion platforms in real-time stream processing pipelines* [Master's thesis, Mälardalen University]. https://urn.fi/URN:NBN:fi-fe2020081048285

[15] Easwaramoorthy, S. V., Yun Xuan, K. O., Putra, L., Ern, N. C., & Sheng, T. J. (2025, January). Comparative study on Oracle, Neo4J, Cassandra, Redis, and MongoDB. *Information Research Communications*, *1*(2), 104–119. https://doi.org/10.5530/irc.1.2.13

[16] Renaldi, R., Santoso, B. C., & Natasya, Y. (2020). Tinjauan pustaka sistematis terhadap basis data MongoDB. *Jurnal Inovasi Informatika*, *5*(2), 132-142.

[17] Levin, S. M. (2024, April). Unleashing real-time analytics: A comparative study of in-memory computing vs. traditional disk-based systems. *Brazilian Journal of Science*, *3*(5), 30–39. https://doi.org/10.14295/bjs.v3i5.553

[18] Pandey, R. (2020). Performance benchmarking and comparison of cloud-based databases MongoDB (NoSQL) vs MySQL (Relational) using YCSB. *Nat. College Ireland, Dublin, Ireland, Tech. Rep*. https://doi.org/10.13140/RG.2.2.10789.32484

[19] Nuriev, M., Zaripova, R., Yanova, O., Koshkina, I., & Chupaev, A. (2024, June). Enhancing MongoDB query performance through index optimization. In *E3S Web of Conferences* (Vol. 531, Article 03022). EDP Sciences. https://doi.org/10.1051/e3sconf/202453103022

[20] Zhu, Y., Xia, T., Zhu, T., Zhao, Z., Li, K., & Hu, X. (2025). RAPO: An Automated Performance Optimization Tool for Redis Clusters in Distributed Storage Metadata Management. *IEEE Access*. https://doi.org/10.1109/ACCESS.2025.3556240

[21] Al Maamari, S. R. S., & Nasar, M. (2025, April). A comparative analysis of NoSQL and SQL databases: Performance, consistency, and suitability for modern applications with a focus on IoT. *East Journal of Computer Science*, *1*(2), 10–15. https://doi.org/10.63496/ejcs.Vol1.Iss2.76

[22] Mohan, R. K., Kanmani, R. R. S., Ganesan, K. A., & Ramasubramanian, N. (2024). Evaluating nosql databases for olap workloads: A benchmarking study of mongodb, redis, kudu and arangodb. *arXiv preprint arXiv:2405.17731*.