Scaling Out with Microservices: A Database-Centric Approach to Monolithic Application Decomposition

Nicolae Mihalache
n.mihalache@student.utwente.nl
University of Twente
The Netherlands

ABSTRACT

In today's fast-paced technological landscape, businesses are constantly looking for ways to scale and remain competitive. One approach that has helped companies is the use of microservices to break down monolithic applications into smaller, more manageable components. This research has the purpose to explore and identify potential problems and best practices when it comes to dividing a monolith app into microservices based on the database schema. The research is expected to provide a quantitative analysis that will compare the monolith and the microservice's performance in terms of scalability and overall response time.

KEYWORDS

Microservices, Monolith, Database schema, Scalability

1 INTRODUCTION

The advancement of technology in recent years has placed significant demands on software development processes and architectures. Organizations are under increasing pressure to deliver software that can scale and evolve rapidly to meet ever-changing business requirements. Monolithic applications, which have long been the prevalent architectural style, are increasingly proving to be a hindrance to agility, flexibility, and scalability, thus negatively affecting overall software development productivity and quality. Microservices architecture has emerged as a promising alternative to monolithic applications, providing multiple benefits such as improved scalability, flexibility, and maintainability [5, 7, 9, 11]. Transitioning from a monolithic to a microservices architecture is a hard task to perform, as it involves careful planning, analysis, and execution to ensure a successful outcome [8]. On the other hand, by decoupling functionality and limiting technical debt, it can streamline the development process and make it easier to introduce new features and automate testing [11].

Existing research has predominantly focused on decomposing monolithic applications based on their functional components or features [3, 6, 8]. While this approach has yielded positive results, there is still a knowledge gap in the literature concerning the implications of a database schema-driven approach to monolithic application decomposition.

39th Twente Student Conference on IT , July 7, 2023,

This research aims to investigate the impact of a database schema-centric approach on the decomposition of monolithic applications into microservices, specifically focusing on the resulting architecture's scalability and performance. By exploring the benefits and drawbacks of this approach, the study seeks to contribute to a more comprehensive understanding of the various factors that should be considered when transitioning from a monolithic to a microservices architecture.

In addition, this study aims to identify best practices and strategies for effectively splitting a monolithic application into microservices based on its database schema, shedding light on practical considerations and challenges that developers and architects may encounter during the decomposition process.

The subsequent sections are arranged in the following manner:

- Section 5 analyses the differences between relational and no-relational scaling databases via a literature overview.
- Section 6 analyses what are the advantages and disadvantegs from partitioning a monolith database into separete microservices databases.
- Section 7 details the performance testing of MySQL and Apache Cassandra through various query and operation tests, by executing a Go script and studying the results in a graphical representation. In this section we also discuss how metrics are collected for both databases and how exactly we can scale Apache Cassandra horizontally.

2 PROBLEM STATEMENT

Even though there has been research conducted on transitioning monolithic applications to microservices, none of them analyzed the impact of partitioning the monolithic database into different microservices that have their own database. This paper will explore how the splitting affects the microservices architecture and how other NoSQL databases can support the microservices architecture.

We have also identified the best practices in the process and how we should split the monolith in microservices based on the database as the primary focus.

2.1 Research Questions

The problem statement led to the following research question: What are the benefits and drawbacks of using a database-centric approach to decompose a monolithic application into microservices?

This has been answered with the following sub-questions:

- How does the database scale in a monolith and microservice architecture?
- What are the advantages and disadvantages of partitioning a relational database?

- What impact does the database splitting have on the query response time?

3 RELATED WORK

The migration from monolithic to microservices-based architectures has attracted significant attention from researchers in recent years. Ivanov and Tasheva [3] introduced a hot decomposition procedure for transitioning operational monolith systems to microservices, addressing challenges such as data consistency and service dependencies. Their approach minimizes downtime during migration and offers valuable insights into effective decomposition strategies.

Milić and Makajić-Nikolić [5] developed a quality-based model for optimizing software architecture and compared monolith and microservice structures. Their study presents a comprehensive analysis of software quality attributes, such as maintainability, scalability, and reliability.

Mparmpoutis and Kakarontzas [6] conducted a mapping study on the use of legacy application database schemas to identify microservices. Their research offers insights into the importance of database schema in microservices decomposition and highlights the potential advantages of a data-centric approach.

Prasandy et al. [8] presented a case study on migrating a monolithic application to a microservices architecture. Their research detailed the challenges encountered during the migration process and evaluated the resulting microservices architecture based on performance and maintainability. This study contributes with valuable practical knowledge on the migration process.

In addition to these studies, Newman [7] provided a comprehensive guide to building microservices, discussing the design principles, best practices, and challenges of adopting this architectural style. Richardson [9] also contributed to the understanding of microservices by offering a set of patterns to help developers design, deploy, and scale microservices-based applications.

4 METHODOLOGIES

To investigate the database-centric approach to monolithic application decomposition, a mixed-methods approach has been employed. First, a systematic literature review was conducted to identify differences between relational databases and no-SQL databases in order to find the potential benefits of no-SQL databases for supporting microservices architecture.

Finally, we delve into a case study that involves a monolithic database. Specifically, we have extracted a portion of the monolith database and transferred it to a microservices database. Our aim has been to analyze the advantages of this move on the microservices architecture and identify the impact it has on query speed.

5 DATABASE SCALING

Monolith applications typically rely on one relationship database, which often becomes the bottleneck of the architecture. A benchmark study [1] found out that "microservices scale better than monolith", and one of the reasons is the database bottleneck. Relational databases are not designed to scale out horizontally, and in order to do so, developers must implement a technique called sharding to partition the database over different servers. However, a recent study [10] found that relational databases perform best on single servers. The second scaling option is adding more CPU and RAM power, but this can become costly very quickly [7].

On the other hand, No-SQL databases have the ability to scale horizontally, which is a major concern of their architecture. Recent research [2, 4] has shown that No-SQL databases, such as MongoDB and Cassandra, scale better than relational databases, such as MySQL.

Cassandra is a distributed No-SQL database that can partition over different servers. In terms of microservices architecture, partitioning can be done so that the microservice and the database partition are coupled on the same server. This achieves a lower response time since the database and the microservice are physically on the same server. This can be achieved by using a Kubernetes infrastructure, which is currently one of the most popular tools for managing containers, according to the 2022 CNCF Survey.

Another issue with monolith applications is that relational databases typically use only one server instance, which increases the round trip time (RTT) to clients that are far away. To replicate it across different servers, a master-slave architecture should be implemented in an ACID manner, which decreases the response time.

However, Cassandra has multiple nodes with a masterless architecture, meaning that any node in the database can provide the same functionality as any other node. This makes it easy to replicate on different servers and reduces the response time by moving servers closer to the users (Figure 1). It is worth noting that Cassandra, along with other No-SQL databases, lacks the ACID properties commonly found in traditional relational databases.

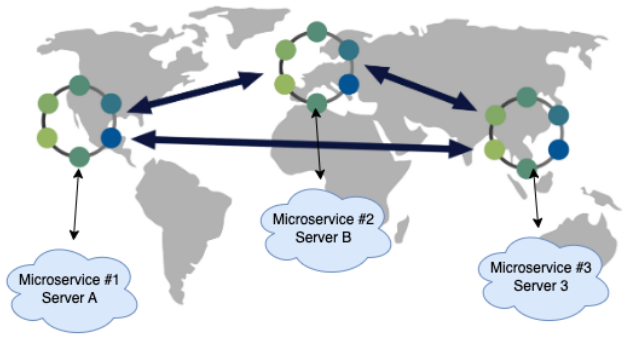

Figure 1: Microservice architecture with Cassandra replication

6 ADVANTAGES AND DISADVANTAGES OF PARTITIONING A RELATIONAL DATABASE

During the transition from a monolithic to a microservices architecture, development teams must carefully contemplate whether to partition the existing database or allocate a new one for each individual microservice, taking into account project-specific requirements and the accumulated technical debt.

6.1 Advantages

The migration from a monolithic to a microservices architecture offers a significant advantage in that it allows development teams

to reset the technical debt associated with database architecture choices. By leveraging accumulated requirements and replacing the relational database with a more suitable alternative within the microservice scope, developers can effectively start with a clean slate, free from the constraints and limitations imposed by prior decisions. This approach requires careful consideration, however, as the decision to partition an existing database or allocate a new one for each microservice must be based on project-specific requirements and technical considerations. As an example, a particular component of the monolithic application that is exclusively used for text searching purposes can be extracted to a microservice that has an elastic search database, thereby enhancing the complexity of the overall architecture but improving response times.

Another great improvement is choosing a distributed database that can replicate across different server locations, which brings the database close to the server, thus decreasing overall RTT.

6.2 Disadvantages

Despite the significant performance benefits that can be achieved through transitioning from a monolithic to a microservices architecture, there are trade-offs to consider. One of the primary challenges that companies face after implementing microservices is the increased complexity of managing and monitoring multiple databases, as opposed to a single database in a monolithic architecture. This can be particularly costly for smaller companies. Additionally, the adoption of microservices requires a skilled development team with a deep understanding of the domain, which may pose a challenge for some companies. Implementing ACID behaviour is another big challenge that developers would have to consider when migrating to a microservice architecture.

7 CASE STUDY

This section presents a comprehensive analysis of the migration process involved in transitioning a section of an app's database from a traditional SQL database to a NoSQL database. This analysis delves into the details of testing and evaluating the performance of the newly implemented database structure.

The initial phase of the assessment involved the identification of a monolithic project that operates on a primary relational database. After extensive research and deliberation, Akaunting [1] was selected as an appropriate candidate for this purpose. Akaunting is a PHP-based project that utilizes the Laravel framework and is equipped with an Object-Relational Mapping (ORM) tool that enables connection to various SQL databases.

The project was selected due to its MySQL monolith database with multiple relationships. In order to get the project up and running, the first step was to create a MySQL instance and follow the Akaunting project GitHub repository instructions.

7.1 Benchmarking

To evaluate the potential of the chosen databases, we created independent instances of MySQL and Cassandra within distinct Docker containers. MySQL was configured as a single instance, whereas Cassandra was configured to take advantage of its capacity to form a cluster with multiple instances.

[1] github.com/akaunting

We chose to operate Cassandra within a three-node cluster for the purposes of this study in order to better comprehend and investigate the potential benefits that could be realized from its scalability when employed in a multi-node configuration (Figure 2).

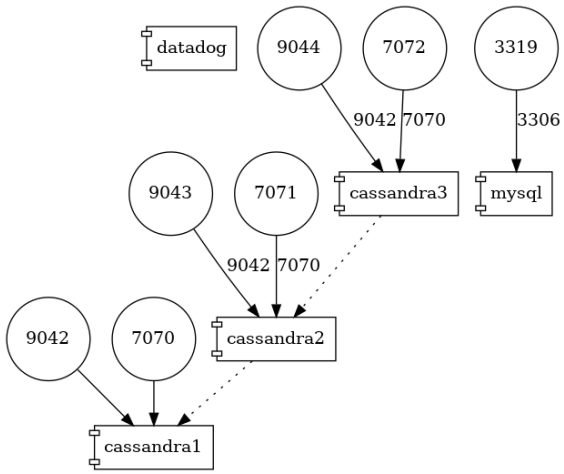

Figure 2: Docker compose containers

Following this, we conducted a microservice extraction analysis, concentrating on the relational database and associated codebase, which revealed the existence of a company document management system. Companies, Users, Documents, Document Totals, and Document Items are the entities that the system supports (Figure 3).

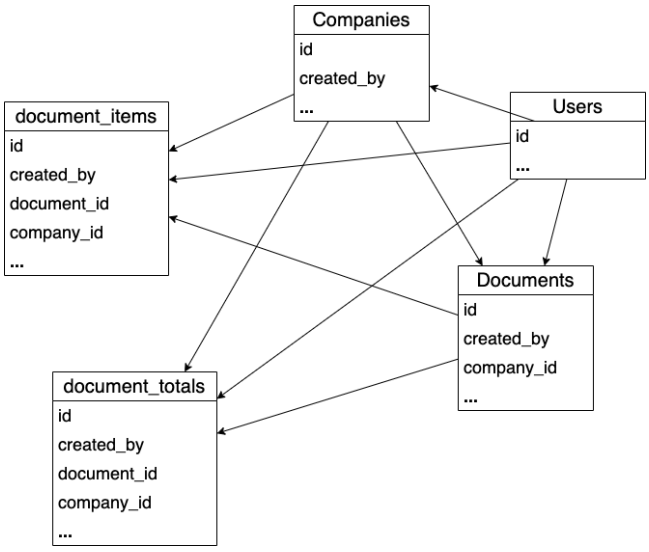

Figure 3: Selected tables from monolith database to be grouped in a microservice

Cassandra is a query-first database. The process of migrating from a relational database to Cassandra requires an initial discernment of the queries we wish to support. Our analysis begins with the following SQL queries:

```
1. SELECT avg(amount) FROM akaunting.documents;
2. Select avg(amount) from akaunting.documents left join
     companies on companies.id = documents.company_id where
     domain ='ss';
```

The first SQL query has no relationships, whereas the second query selects all average quantities by the company domain by doing a join. Cassandra employs partition keys; each partition key is used to improve data placement, thereby enhancing the efficiency of data writing and reading. Given the specific nature of our query, which focuses on documents categorised by the company id, it makes sense to designate the company id as our partition key for the first query. In an effort to replicate the functionality of the MySQL database, we propose incorporating the document identifier as the secondary key. This strategy enables the ordering of documents by id within each partition and the execution of queries by id, thereby substantially enhancing the overall efficiency and effectiveness of our database.

```
CREATE TABLE IF NOT EXISTS akaunting.documentsByCompanies
     (id int, company_id int, document_number text, type
     text, status text, amount double, PRIMARY KEY
     (company_id, id));
```

For the second query however, we created a separate Cassandra table, where the domain of the company is the partition id.

```
CREATE TABLE IF NOT EXISTS
     akaunting.documentsByCompaniesDomain (id int, domain
     text, document_number text, type text, status text,
     amount double, PRIMARY KEY ((domain), id)) WITH
     CLUSTERING ORDER BY (id DESC);
```

The Akaunting project utilizes seeding scripts to generate random data pertinent to the entities selected for our research. An integral part of this process required the creation of migration scripts which could effectively map MySQL queries onto Cassandra's table structure.

Using two distinct methods, metrics were gathered. Initially, a database was created solely for the purpose of aggregating metrics. Therefore, each time a query was executed, its duration was recorded in this specialized database. A script written in the Golang programming language was used to initiate the testing procedure for metric collection. This script initiated multiple concurrent green threads, each of which performed a single read or write operation. Each query's execution time was subsequently logged in the metrics database.

Testing the first query: Initially, there were approximately 3000 entries in the database. A Go script was executed with a range of coroutines, including 10, 50, 100, 150, and 350, each coroutine

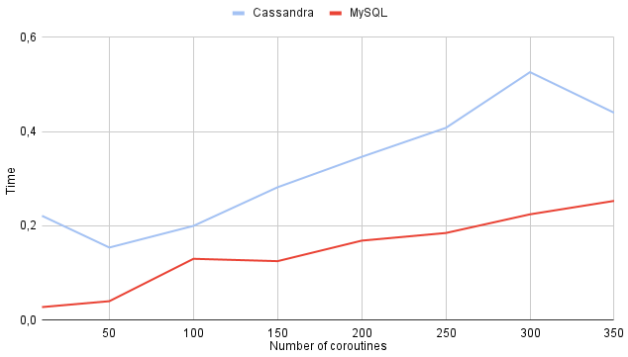

Figure 4: Read test 1st query (Time of execution / Number of green threads)

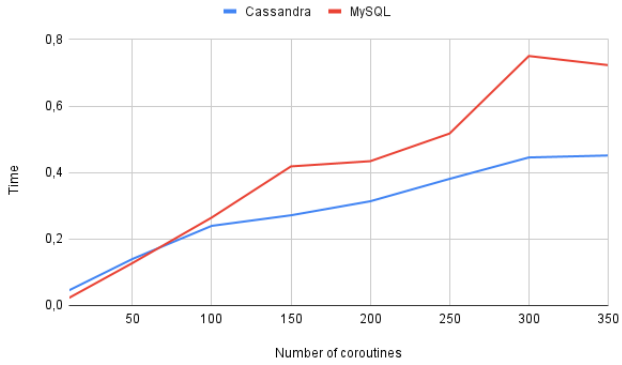

Figure 5: Read test 2nd query (Time of execution / Number of green threads)

was executing a database query and coroutines are executed in a concurrent manner. Figure 4 is a graphical representation of the average query time for each worker count. Figure 4 shows that MySQL's read performance exceeded that of Cassandra.

Testing the second query: After we examined the results from Figure 5, we noticed that using joins in SQL is slower than getting data from just one table. This makes sense because a join makes the query look in two tables, not just one. On the other hand, Cassandra, which adopts a 'query first' approach, organizes the requisite data across different partitions and allocates one table structure for each query. While this method does consume additional storage space and demands higher maintenance effort from the project developers, its efficiency cannot be underestimated.

In the next phase of our investigation, we analyzed the efficiency of insert queries in each database system. Figure 6 shows a noteworthy pattern: as the number of concurrent workers executing insert operations increased, the performance of MySQL exhibited a discernible decline, with insert operations gradually decelerating. This trend was not reflected in Cassandra's case. In contrast, the time required for insert operations in Cassandra remained constant

regardless of the number of concurrent inserting workers. This result suggests that Cassandra may have an advantage over MySQL in scenarios requiring a large number of concurrent insert operations.

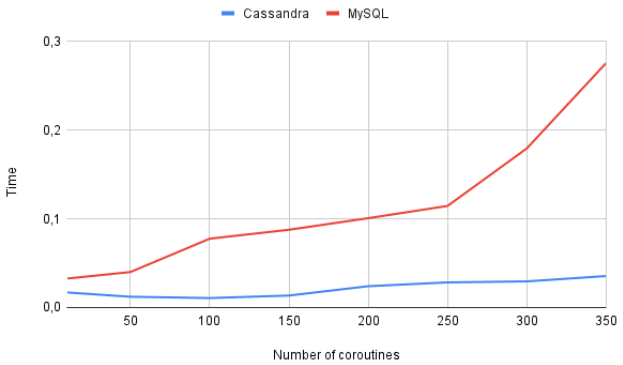

Figure 6: Write test (Time of execution / Number of green threads)

In the final phase of our investigation, we conducted a test in which the Go script invoked an equal number of read and write operations simultaneously. Figure 7 revealed a significant performance gap between the two database management systems. In this scenario, MySQL demonstrated inferior performance. A plausible explanation for this is that MySQL has additional functionality to preserve, specifically, ACID properties (Atomicity, Consistency, Isolation, and Durability). While assuring transaction reliability and integrity, these properties could potentially impose a performance burden on write operations, thereby slowing down read operations.

Cassandra, on the other hand, demonstrated superior scalability for concurrent inserts and writes. This may be due to its eventual consistency model, which permits a greater write throughput by relaxing the requirement for immediate consistency. These results suggest that the choice between MySQL and Cassandra may hinge on the application's specific requirements, specifically the need for ACID compliance versus the need for high write throughput. Further research is required to investigate these tradeoffs in greater depth.

MySQL outperforms Cassandra in terms of read performance, in accordance with our findings and the findings of other benchmarking studies. On the other hand, Cassandra appears to outperform MySQL for insert operations. These results suggest that the application's specific operational requirements should govern the choice between MySQL and Cassandra. [12]

7.2 Metrics

The possibility of observing database errors and the accumulation of metrics are essential capabilities for database management and oversight. This section describes the implementation of metric collection for each database and describes the various categories of available metrics.

This thesis' scope of investigation extended beyond manual metric collection to include automated metric analysis. A suite of data analytics tools was applied to analyze the compiled metrics from

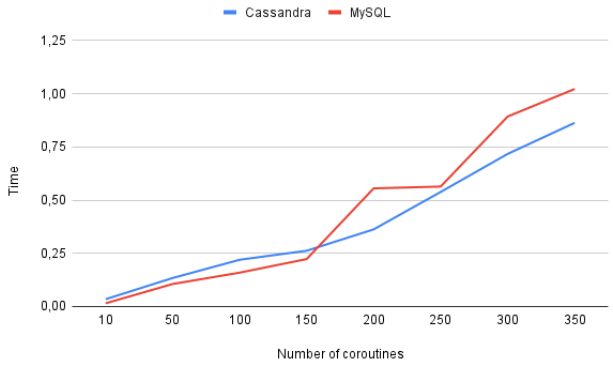

Figure 7: Concurrent reads and writes test (Time of execution / Number of green threads)

the respective databases. Datadog[2] was utilized specifically for the analysis of MySQL metrics data. Grafana[3], the Grafana Agent, and Prometheus[4] were utilized concurrently to analyze metrics from both the Cassandra cluster.

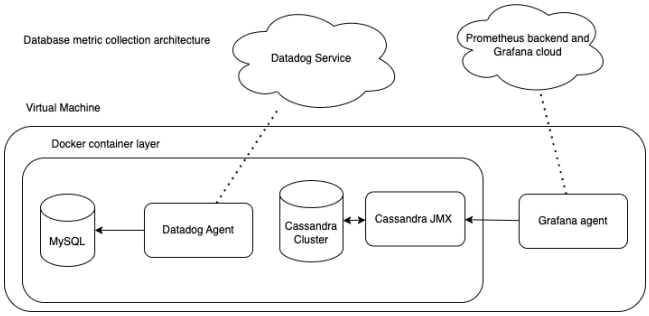

Figure 8: Database metric collection architecture

Comparing the monitoring systems Cassandra and MySQL reveals several significant distinctions. In terms of available metrics, Cassandra provides a broader selection of metrics for agent collection from the outset compared to MySQL. This contains valuable metrics pertaining to writing and reading latencies, including computed percentiles.

Implementing equivalent metrics in MySQL requires additional development effort, which may pose difficulties and influence the process's efficiency.

Cassandra's design architecture contributes considerably to the accessibility of its metrics. Each node in a Cassandra cluster communicates with the Grafana agent and provides abundant granular data. This inter-node communication enables the collection of exhaustive metrics, providing greater insight into the operation and performance of the system.

[2]https://www.datadoghq.com/
[3]https://grafana.com/products/cloud/
[4]https://prometheus.io/

In contrast, when collecting metrics for MySQL, the Datadog agent queries the Performance Schema. However, the MySQL Performance Schema does not expose as much information as a Cassandra node, resulting in a smaller range and depth of monitoring-accessible data.

Regarding the availability of metrics, Cassandra appears to offer more advantages than MySQL. This is not to imply that MySQL is inferior, but it does highlight the distinctions between the two systems' design philosophies and capabilities. Depending on the requirements and resources of a project, these distinctions could have a significant impact on the selection of a monitoring system.

7.3 Scalability

In the section 7.1, we highlighted that scaling MySQL across multiple servers presents a significant challenge. Performance analysis revealed that while MySQL performs excellently for read-intensive operations, it may not be the best choice when the application necessitates storing large amounts of data that exceed the capacity of a single server and has a high volume of write requests.

Under these conditions, Cassandra emerges as a more suitable choice due to its ability to scale by distributing different nodes across multiple virtual machines, thus achieving horizontal scaling. This discussion illustrates an example of how to horizontal scale Cassandra nodes in a Docker Compose environment.

The environment configuration described here can be easily adapted for use with Docker Swarm, Kubernetes, or other container orchestration systems. This flexibility enhances the versatility of Cassandra and further illustrates its advantages in scenarios that demand high scalability.

Figure 2 provides a clear visualization of the Docker Compose system architecture. Despite its seemingly simplistic nature, Cassandra performs a variety of complex operations beneath the surface. For instance, Cassandra nodes discover each other through a process of communication utilizing the gossip protocol.

Once all the Cassandra nodes have been identified and a masterless cluster has been formed, keyspaces can be created. A keyspace in a Cassandra cluster is the highest-level data container that controls how data is replicated across nodes. Keyspaces can employ various replication strategies across the cluster to suit different requirements.

To scale Cassandra nodes across multiple data racks, the use of the NetworkTopologyStrategy is advised. This strategy aims to distribute replicas across different racks as nodes within the same rack (or a similar physical grouping) often experience concurrent failures due to power, cooling, or network issues.

The following Cassandra command shows how to create a keyspace that employs this strategy. The 'class' indicates the desired strategy, while the 'replication_factor' specifies the number of copies of each row to be made within the data cluster:

```
CREATE KEYSPACE IF NOT EXISTS akaunting WITH REPLICATION =
    { 'class' : 'NetworkTopologyStrategy',
    'replication_factor' : '3' };
```

This command creates a keyspace named "akaunting," if it does not already exist, and sets the replication strategy to NetworkTopologyStrategy with a replication factor of three. This means that three copies of each row will be created in the data cluster, providing redundancy and resilience.

To execute a Cassandra cluster in a Docker Compose environment, a new Cassandra image was locally created, which employs the jmx_prometheus_javaagent[5] to expose an endpoint for metric collections. Assume that the image is named 'mycassandra:v2'.

```
cassandra1:
 image: mycassandra:v2
 ports:
  - 9042:9042
  - 7070:7070
 volumes:
  - cassandra_data_1:/var/lib/cassandra
 environment:
  - CASSANDRA_CLUSTER_NAME=Test Cluster
  - CASSANDRA_SEEDS=cassandra1,cassandra2
  - CASSANDRA_DC=se1
  - CASSANDRA_ENDPOINT_SNITCH=gossip
cassandra2:
...
cassandra3:
...
```

In this configuration, the following parameters are defined:

- **image:** Specifies the Docker image to use for the container, in this case, 'mycassandra:v2'.
- **ports:** Maps the container's ports to the host's ports. Here, 9042 is the Cassandra client port and 7070 is for metric collections.
- **volumes:** Maps the Docker volume 'cassandra_data_1' to the path '/var/lib/cassandra' in the container. This is where Cassandra's data files will be stored.
- **environment:** Sets various environment variables for the container:
  - **CASSANDRA_CLUSTER_NAME:** Specifies the name of the Cassandra cluster.
  - **CASSANDRA_SEEDS:** Provides a comma-separated list of seed nodes used when a new node joins the cluster.
  - **CASSANDRA_DC:** Identifies the data center to which the node belongs.
  - **CASSANDRA_ENDPOINT_SNITCH:** Sets the endpoint snitch, which tells Cassandra about the network topology to route requests efficiently.

The configurations for 'cassandra2' and 'cassandra3' follow the same structure, with adjustments as needed to accommodate their specific roles and settings in the cluster.

The ease of horizontal scaling with Apache Cassandra, particularly in a containerized environment, stands in stark contrast with the complexity that characterizes the scaling process for MySQL. The fundamental difference lies in their respective architectures: while Cassandra is designed with a distributed system in mind,

[5]https://github.com/prometheus/jmx_exporter

MySQL's design revolves around a single-node concept. This absence of built-in distributed system capabilities in MySQL makes horizontal scaling a more involved process.

To achieve horizontal scaling with MySQL, a typical approach is to establish replication or clustering, each of which comes with its own complexity that demands careful planning, detailed configuration, and ongoing maintenance. Replication entails the creation of additional MySQL servers that serve as 'slaves' to the 'master' server. The master server processes all write operations, whereas read operations are distributed among the slave servers. Conversely, clustering involves the creation of a group of servers that operate as a single server. In applications, both strategies require considerable effort and technical expertise to be implemented correctly.

It is thus evident that although both Apache Cassandra and MySQL have the potential for horizontal scaling, the process is significantly simpler and more straightforward with Cassandra due to its inherent distributed features. Despite the complexity associated with scaling MySQL, when properly configured and maintained, both systems are capable of effectively managing large-scale data tasks

8 CONCLUSION

In conclusion, the comparative analysis of MySQL and Apache Cassandra in this thesis demonstrates that the choice between these two database management systems depends significantly on the operational requirements of a particular application. The results demonstrated that MySQL excels in read performance but may struggle with a large volume of concurrent write operations, particularly when scaled across multiple servers.

In contrast, Cassandra excels in terms of write and insert performance, especially under high concurrency. Its distributed design allows for smoother and simpler horizontal scaling compared to MySQL, which should be considered when developing applications that are intended to process massive amounts of data or require high write throughput.

However, it is also essential to keep in mind that MySQL provides robust transaction reliability and integrity through its ACID properties, which could be crucial for applications requiring immediate data consistency. In contrast, Cassandra's eventual consistency model allows for a higher write throughput but may not be viable for use cases that require instant data consistency.

Regarding metric collection and monitoring, Cassandra seems to offer more out-of-the-box compared to MySQL, which could impact the efficiency of maintaining and observing the database. However, these metrics' richness and value would still be subject to the specifics of an application's requirements and the resources allocated to manage them.

MySQL and Apache Cassandra each have advantages and disadvantages, but the decision should always be based on the specific requirements of the application, the available resources, and the context in which these systems will be used. Future research could investigate the trade-offs between these two systems in a variety of application scenarios in order to provide more nuanced insights to developers and organizations seeking to make informed decisions about their database management systems.

REFERENCES

[1] Nicholas Bjørndal, Manuel Mazzara, Antonio Bucchiarone, Nicola Dragoni, and Schahram Dustdar. 2021. Migration from Monolith to Microservices: Benchmarking a Case Study. *The Journal of Object Technology* (2021). https://doi.org/10.5381/jot.2021.20.2.a3

[2] Ticiana Capris, Pedro Melo, N. Garcia, I. Pires, and Eftim Zdravevski. 2022. Comparison of SQL and NoSQL databases with different workloads: MongoDB vs MySQL evaluation. *2022 International Conference on Data Analytics for Business and Industry (ICDABI)* (2022). https://doi.org/10.1109/icdabi56818.2022.10041513

[3] Nikolay Ivanov and Antoniya Tasheva. 2021. A Hot Decomposition Procedure: Operational Monolith System to Microservices. *International Conference on Applied Informatics* (2021). https://doi.org/10.1109/icai52893.2021.9639494

[4] Vishal Dilipbhai Jogi and Ashay Sinha. 2016. Performance evaluation of MySQL, Cassandra and HBase for heavy write operation. *International Conference on Recent Advances in Information Technology* (2016). https://doi.org/10.1109/rait.2016.7507964

[5] Miloš Milić and Dragana Makajić-Nikolić. 2022. Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures. *Symmetry* (2022). https://doi.org/10.3390/sym14091824

[6] Antonios Mparmpoutis and George Kakarontzas. 2022. Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study. *International Conference on Advancements in Computational Sciences* (2022). https://doi.org/10.1145/3564982.3564995

[7] S. Newman. 2015. Building microservices - designing fine-grained systems, 1st Edition. *null* (2015).

[8] Teguh Prasandy, Dina Fitria Murad, and Taufik Darwis. 2020. Migrating Application from Monolith to Microservices. *International Conference on Information Management and Technology* (2020). https://doi.org/10.1109/icimtech50083.2020.9211252

[9] C. Richardson. 2018. *Microservices Patterns: With Examples in Java.*

[10] Samidi Samidi, Ronal Yulyanto Suladi, and Ario Bambang Lesmana. 2022. Implementation of Database Distributed Sharding Horizontal Partition in MySQL. Case Study of Application of Food Serving On Kemkes. *Jurnal Sisfotek Global* (2022). https://doi.org/10.38101/sisfotek.v12i1.477

[11] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. 2017. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *null* (2017). https://doi.org/10.1109/mcc.2017.4250931

[12] Piush Vaish. 2023. *A Comparison between Cassandra and MySQL.* https://adataanalyst.com/data-analysis-resources/a-comparison-between-cassandra-and-mysql/