INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Tradeoffs entre Desempenho e Energia em Aplicações Java entre Bases de Dados Relacionais e Não Relacionais

ROGÉRIO MOURA DE SOUSA
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Performance-Energy Tradeoffs in Java Applications between Relational and Non-Relational Databases

Rogério Moura de Sousa

Dissertation

Master's Degree in Computer Engineering

Specialisation in Software Engineering

Acknowledgements

This section is dedicated to expressing my heartfelt gratitude to all those who have played a meaningful role in the successful completion of my dissertation.

To my family, I extend my deepest thanks for all the support and encouragement. Your belief in me has been a cornerstone of my academic journey.

I am grateful to my supervisor at ISEP, Teacher Isabel Azevedo, who guided me during the resolution of this dissertation and was available for my doubts and to give me crucial feedback for the successful work.

I appreciate my Academic Institution, ISEP, for the insights given that have contributed to my personal and academic development. It was essential to perform this project.

I appreciate the companionship of my colleagues and closest friends; you have given me the required motivation throughout this journey.

Also, it was required resilience, discipline, and determination to complete this project under such demanding and adverse circumstances required which allows me to acknowledge myself. I am proud of the perseverance I showed and of having brought this work to completion despite the odds.

Declaration of Integrity

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section “Ethical considerations” of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, April 21, 2025

Resumo

Esta dissertação investiga os compromissos da performance e consumo de energia em aplicações Java que utilizam tecnologias de bases de dados relacionais e não relacionais. Com o aumento da complexidade e do volume de dados, a seleção de um sistema de base de dados adequado tornou-se uma decisão crítica para os programadores e arquitetos de sistemas. As bases de dados relacionais, como o PostgreSQL, destacam-se pelo bom desempenho na manipulação de dados organizados e em operações de consulta sofisticadas. Ao contrário das não relacionais como o MongoDB e o Apache Cassandra que são mais indicadas para contextos que exigem elevada escalabilidade e maior adaptabilidade no tratamento de grandes volumes de dados ou informações com estrutura menos definida.

O estudo adota uma metodologia rigorosa que combina uma revisão da literatura e experiências controladas para analisar três aspetos fundamentais: o impacto do tipo de base de dados no desempenho e no consumo de energia em diferentes operações da base de dados, os benefícios das técnicas de otimização de consultas, como a indexação, e o papel do volume de dados na eficiência do sistema. Foram concebidos e executados vários cenários de teste utilizando ferramentas como o Grafana k6 e o Kepler para medir métricas de desempenho e energia relevantes.

Os resultados confirmam que não existe uma solução única para todos os casos. Embora as bases de dados relacionais tenham um melhor desempenho em operações de consulta com um volume de dados elevado e cargas de trabalho de consulta estruturadas, os sistemas não relacionais oferecem escalabilidade e capacidade de resposta superiores em ambientes de dados de alto rendimento ou em grande escala. Além disso, foi observado que a utilização de estratégias de otimização melhora significativamente a eficiência energética e o desempenho em todos os tipos de bases de dados. Estas conclusões sublinham a importância de alinhar as tecnologias de bases de dados com os requisitos específicos das aplicações para obter soluções otimizadas em termos de energia e de desempenho.

Palavras-chave: Java, Performance, Energia, Bases de Dados Relacionais, Bases de Dados não relacionais, SQL, NoSQL

Abstract

This dissertation investigates the performance and energy consumption tradeoffs in Java applications using relational and non-relational database technologies. With the increasing complexity and volume of data, the selection of an appropriate database system has become a critical decision for developers and system architects. Traditional relational databases such as PostgreSQL are known for their reliability and efficiency in structured data and complex queries, while non-relational databases like MongoDB and Apache Cassandra provide greater flexibility and scalability for handling unstructured or large datasets.

The study adopts a rigorous methodology combining a literature review and controlled experiments to analyse three core aspects: the impact of the database type on performance and energy consumption in different database operations, the benefits of query optimisation techniques such as indexing, and the role of data volume in system efficiency. Multiple test scenarios were designed and executed using tools such as Grafana k6 and Kepler to measure relevant performance and energy metrics.

The results confirm that no single solution fits all cases. While relational databases perform better in consistency-heavy select operations and structured query workloads, non-relational systems offer superior scalability and responsiveness in high-throughput or large-scale data environments. Additionally, the use of optimisation strategies was shown to significantly improve both energy efficiency and performance across database types. These findings highlight the importance of aligning database technologies with application-specific requirements to achieve energy-aware and performance-optimised solutions.

Keywords: Java, Performance, Energy, Relational Databases, Non-relational Databases, SQL, NoSQL

Table of Contents

List of Figures

List of Tables

List of Code Snippets

Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DBMS	Database Management System
DDL	Data Definition Language
HQL	Hibernate Query Language
JDBC	Java Database Connectivity
JPA	Jakarta Persistence API
JSON	JavaScript Object Notation
NoSQL	Not Only SQL
ORM	Object-Relational Mapping
POJO	Plain Old Java Objects
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
RDBMS	Relational Database Management System
SQL	Structured Query Language
VDS	Virtual Database System
WBS	Work Breakdown Structure
EPA	Environmental Protection Agencies
GQM	Goal Question Metric
VM	Virtual Machine

1 Introduction

This chapter introduces the study's context and then describes the problem and its objectives, available in sections 1.1, 1.2 and 1.3, respectively. Finally, the ethical considerations of the work and the structure of the document are explained in sections 1.4 and 1.5.

1.1 Context

Companies like Google, Facebook, and Twitter require storing a large amount of data in today's applications. It is applied to unstructured data, which presents challenges for its storage. While relational databases are effective for handling structured data, they are not designed to manage unstructured formats. This limitation has led to the growing adoption of NoSQL database management systems, specifically designed to process and store unstructured data efficiently.

For example, Slovak Railways maintains an extensive database that maps connections between cities and villages across Slovakia [1]. When this database was initially created, only relational database management systems such as Oracle, MySQL, and Microsoft SQL Server were available. However, as data requirements have evolved, modern databases must now address critical factors like scalability, performance, and latency to ensure user satisfaction [1].

Social media platforms like Facebook and Google handle massive user traffic and require database solutions capable of processing large datasets with high-frequency read and write operations. To meet these demands, platforms have transitioned from traditional relational databases to NoSQL systems like Apache Cassandra and Google BigTable. These NoSQL databases offer the flexibility and efficiency needed to manage the challenges of modern, high-traffic applications [2].

Given the rapid growth in data complexity, understanding existing database systems is crucial for making informed decisions. Innovations often build upon what already exists, emphasising the importance of studying current technologies to select the best solutions for specific use cases.

1.2 Problem description

There has been a growing concern about reducing energy consumption in software product development to reduce financial costs and heat emissions, and support a green environment [3].

Some studies [4] and [5] mention that performance and energy consumption are correlated, the second study specifies the Java application experiments. Increased performance often leads to higher energy consumption due to the more intensive usage of computing resources. For example, in the analysis of NoSQL databases, higher workload scenarios often demand more CPU and memory, leading to a proportional increase in energy consumption. In databases, energy consumption is directly tied to latency and throughput; optimising for one may lead to decreased results in the other. For instance, non-relational databases can save energy by reducing the power allocated to idle states or synchronising I/O operations, but these measures may increase latency. Both studies highlight instances where energy efficiency does not scale linearly with performance improvements. For example, an experiment showed that performance optimisation in MongoDB and Cassandra doesn't always lead to proportional energy savings due to factors like I/O bottlenecks and poor synchronisation related to these databases [4].

Performance can be influenced in several ways:

1. Analyse the best approach of an ORM or JDBC for the application [5].
2. Choosing the best option of relational or non-relational databases [6].

A recent study [4] concluded that previous performance and energy comparisons between relational and non-relational databases were made for a limited category of NoSQL databases, especially document-oriented databases like MongoDB. Also, the impact of query optimisation on energy efficiency has been observed for a few query operations, like simple get operations, inserts, and a few usages of index strategies. The same study determined that data modelling significantly affects query performance, with only one study addressing this issue in its literature review.

One study concluded that there are a lot of differences when studying and comparing the performance and energy consumption between multiple programming languages, and Java is in the top 5 best languages [7]. Also, Java was the top 7 most used programming language in 2024 [8]. Therefore, it is essential to analyse the performance-energy tradeoffs in Java applications when deciding between relational or non-relational databases.

1.3 Objective and Research Methodology

The primary objective of this research is to complement existing studies by analysing the performance and energy consumption tradeoffs, addressing the impact of query optimisation and data volume, between relational and non-relational databases in Java-based applications. This investigation focuses on the impact of query operations involving indexes and varying

query complexities, for example, queries with low JOIN operations against multiple JOIN operations. Also, it analyses how the data volume impacts the performance and energy. Additionally, the study aims to analyse a range of NoSQL databases, MongoDB and Apache Cassandra against SQL databases, MySQL, to provide a comprehensive understanding of their performance and energy efficiency characteristics, as outlined in the project description in 1.2. This research aims to provide system architects and developers with practical guidance for selecting and configuring database systems to achieve energy-conscious and high-performance software solutions by performing controlled experiments.

To achieve these objectives, the research methodology is designed to answer the following research questions by doing a Literature Review:

- **RQ1**: How does the choice between relational and non-relational databases impact Java-based applications' energy consumption and performance?
- **RQ2**: How does query optimisation for relational databases affect energy consumption and performance compared to non-relational databases?
- **RQ3**: How does data volume influence query performance and energy consumption in relational and non-relational databases?

Also, a controlled experiment will be conducted to conclude these objectives, taking into consideration a designed Goal Question Metric and hypothesis tests to analyse the results.

1.4 Ethical considerations

Throughout this research, ethical principles have been carefully adhered to, ensuring integrity, responsibility, and respect for all stakeholders involved. The guidelines followed include:

- Upholding the ACM Code of Ethics by prioritising transparency, fairness, and careful consideration in all decisions, particularly when handling sensitive data or modifying systems [9], [10].
- Adhering to the NSPE Code of Ethics for Engineers by maintaining professional integrity, safeguarding public interest, and ensuring the research contributes positively to societal well-being [11].
- Following the IPP Code of Good Practices and Conduct [12] including the Declaration of Integrity.
- Ensuring the research aligns with the IEEE/ACM Software Engineering Code of Ethics by prioritising the public interest and committing to the delivery of high-quality and efficient products. This includes rigour in all experimental processes and data handling, Clause 3, by upholding the highest standards of accuracy, reliability, and ethical responsibility in data collection, analysis, and reporting [13].

These principles guided every stage of this research, ensuring it aligns with the highest standards of ethical conduct. All the participants were asked for their approval for the Java application usage, which was all positive. This is visible in Appendix B.

Only three databases were chosen in the experiment due to the author's limited time.

It is important to consider that the data models and the experimental decisions made influence the results obtained and conclusions drawn.

1.5 Document Structure

This document is structured into seven chapters: Skills Management and Project Planning, Background, Literature Review, followed by the Experimental Design and Implementation, and, finally, the Analysis of Results and a Conclusion. Also, at the end of the document, there is the necessary appendix that complements the document.

Beginning with Skills Management, where the most developed skills are identified, followed by skills that require further improvement. The Project Planning describes the identified stakeholders, benefits, constraints, assumptions, costs, and risks, accompanied by the project schedule for the development of the dissertation.

The Background chapter provides a comprehensive overview of the theoretical foundations, focusing on relational and non-relational databases, including key-value, column, and document storage types. It also gets into the persistence layer, indicating technologies like JDBC and JPA, and discussing frameworks such as Hibernate and EclipseLink. It introduces the concepts of performance and energy, mentioning metrics used for their measurement and available tools to perform them.

The Literature Review concludes the survey of existing research, outlining the methodology employed, covering research questions, data sources, search strategies, and inclusion criteria, followed by a description of the data collection process. The gathered literature is subsequently analysed, with the discussion structured around the established research questions, offering evidence-based responses to each.

The Design chapter informs about the tools and frameworks used for the investigation, the project chosen, and its architecture, finalising with the defined scenarios that will be executed in the experiments.

The Implementation chapter explains how the project was implemented. Starting with how the databases were created, in this case in the Docker container, followed by the generation of scripts to bulk large amounts of data at once and then explaining how the application connects to the databases. Finally, it is explained how the tools to measure performance and energy consumption were integrated into the application.

Afterwards, there is the Analysis of Results chapter that starts with the methodology used to get conclusions about the results, hypothesis tests, and then the analysis itself, with the performance and energy consumption tradeoffs gathered.

Finally, there is a Conclusion about the investigation performed during this report.

2 Skills Management and Project Planning

This chapter is responsible for showing the Skills Management and Project Planning developed for the successful resolution of this dissertation thesis, in sections 2.1 and 2.2, respectively.

2.1 Skills Management

Skills Management involves identifying strengths and areas for improvement, setting actionable goals, and defining specific steps to enhance personal and professional competencies.

2.1.1 Identification of Required Skills

It is essential to identify the skills required to achieve the outlined objectives for the successful execution of the project. Table 1 provides a comprehensive analysis of the skills necessary, the current proficiency levels, identified gaps, and relevant comments. The skills listed encompass technical and non-technical areas, ensuring a balanced approach to the project requirements.

Skill	Required Proficiency	Current Proficiency	Gap	Comments
Java Language	4	4	Low	Strong Java knowledge is essential.
ORM Frameworks	4	3	Low	A basic understanding of ORM tools; requires practice for fine-tuning relational database performance.
JDBC	4	4	Low	Solid knowledge of JDBC.
SQL Databases	5	4	Low	Proficient in SQL databases.
NoSQL Databases	5	2	High	Limited knowledge of NoSQL; requires deeper exploration.
Database Indexing Techniques	4	2	Medium	Limited experience with indexing strategies; critical for optimising query performance.
Performance Testing	5	4	Low	Experience in testing frameworks; focus on advanced performance profiling tools.
Energy Testing	5	1	High	Minimal understanding of energy testing techniques; critical for evaluating trade-offs in performance.
Communication	4	3	Medium	Effective in written communication; needs improvement in presenting

				technical results to non-technical stakeholders.
Github	3	5	Low	Strong proficiency in GitHub for version control and collaboration; no significant gaps identified.
Professional Experience	4	3	Medium	Practical exposure to industry practices; requires further experience in managing complex, real-world database scenarios.

Table 1 - Required Skills

2.1.2 Assessment of Current Skills

The assessment of current skills is a critical step in understanding the baseline knowledge and capabilities that underpin the success of the project. This topic identifies areas of strength that will serve as a foundation while highlighting gaps that must be addressed.

2.1.2.1 Strengths

Understanding Java concepts such as object-oriented programming and data structures ensures a strong capability to build and optimise application logic. This skill will be crucial in implementing projects to measure and optimise performance and energy trade-offs.

A strong experience with relational databases and skills in technologies like JDBC provide a good experimental approach to managing relational data. These skills are critical to analyse the role of performance in relational databases, especially when comparing them with non-relational ones.

Advanced version control and collaboration skills through GitHub, ensuring that all project artefacts are well-organised and documented. This expertise facilitates seamless collaboration and effective integration of code changes.

2.1.2.2 Gaps and Areas for Development

A significant gap exists in knowledge and practical experience with NoSQL systems. These databases are essential for managing unstructured and semi-structured data, providing scalability and flexibility. The lack of familiarity with NoSQL indexing, query optimisation, and schema-less design principles restricts the ability to make thorough performance comparisons.

Limited experience with database indexing strategies is another challenge. Indexing is essential when studying performance and energy consumption. However, it can have bad results when applied incorrectly. Acquiring in-depth knowledge of indexing will allow the researcher to implement efficient solutions and conduct a more detailed analysis of their impact.

Minimal exposure to energy testing tools and techniques represents a critical gap in skill, since the study involves analysing the performance and energy tradeoffs in Java applications. Also, professional exposure is important to simulate and replicate real case scenarios.

2.1.3 Strategies for Skill Development

This action plan has been developed following the SMART methodology, ensuring that each action is **Specific, Measurable, Achievable, Relevant, and Time-Bound**. By adhering to these principles, the plan aims to facilitate structured and effective personal development:

- **Specific**: Clearly defines the goal to focus efforts on targeted improvements.
- **Measurable**: Includes criteria to track progress and evaluate success.
- **Achievable**: Sets realistic and attainable objectives to maintain motivation.
- **Relevant**: Aligns with broader professional and personal aspirations.
- **Time-Bound**: Establishes deadlines to ensure accountability and progress within a set timeframe.

This approach provides a robust framework for skill development, ensuring consistent progress toward achieving excellence in the identified competencies.

The Table 2 shows actionable steps to be taken to improve the skills to be developed:

Skill	Actions to improve
NoSQL Databases	Enrol in an advanced NoSQL course focusing on schema design, indexing, and query optimisation, completing all modules and exercises within six weeks.
	Attend the Database Administration course and finish it in the first semester.
Database Indexing Techniques	Write a report analysing the impact of indexing strategies on query performance, documenting results for five scenarios, to be completed in four weeks.
	Attend the Database Administration course and finish it in the first semester.
Energy Testing	Learn to use Kepler for energy profiling by conducting five energy measurement sessions on test configurations within six weeks.
	Design a standardised energy testing workflow, including configuration and data collection processes, applying it to three database setups within one month.
ORM Frameworks	Build a small Hibernate project focusing on caching, lazy loading, and query generation, implementing three use cases, to be completed in five weeks.

	Conduct performance tests comparing Hibernate-generated queries with direct SQL, analysing metrics for five scenarios within three weeks.
Communication Skills	Attend a professional writing course and complete all the tasks in 8 weeks.
	Write a weekly article on my field of study or work for the next 3 months.
Professional Experience	Participate in three simulated case studies focused on complex database projects, documenting solutions and results within three months.
	Engage in professional forums or communities (e.g., LinkedIn groups), contributing to five discussions and seeking feedback within two months.

Table 2 - Skills Management Plan

2.1.4 Identification of Skills to Improve

The skills to improve fall into two primary categories: technical skills, which form the foundation for rigorous data analysis and system implementation, and soft skills, which enhance the ability to communicate findings effectively and manage collaborative challenges. Each of these categories is critical to the success of the dissertation and the ability to produce results that are accurate and impactful in both academic and professional contexts.

Based on the analysis, the skills to improve are categorised as follows:

- Technical Skills
    - Advanced techniques in NoSQL databases.
    - Indexing and query optimisation for relational and non-relational systems.
    - Knowledge of energy testing methodologies to assess trade-offs.
    - ORM frameworks experience.
- Soft Skills
    - Communication, particularly in presenting complex technical findings.
    - Professional acumen through simulated or collaborative projects.

2.2 Project Planning

This chapter points out the main elements of the project charter in the section 2.2.1, followed by the scope of work and projected schedule in sections 2.2.2 and 2.2.3. Finally, it presents the costs and the risks involved in sections 2.2.4 and 2.2.5.

2.2.1 Main Elements from Project Charter

The project charter defines the main stakeholders and the benefits, constraints and assumptions. This approach supports a systematic and goal-oriented path toward achieving the research objectives.

2.2.1.1 Main Stakeholders

The success of the project relies on the engagement and influence of multiple stakeholders, divided into main and secondary groups. These individuals or entities are essential in shaping the project's direction, assessing its progress, and ultimately gaining value from its results, thereby ensuring that the work remains pertinent and aligned with both scholarly and practical goals.

Several stakeholders were identified:

- **Primary Stakeholders**:
  - **Developers**: Interested in improved performance and energy efficiency solutions for Java applications.
  - **Organisations**: Focused on the practical application of findings to reduce performance, energy costs and environmental impact.
  - **Thesis Evaluators:** Responsible for evaluating the final dissertation, ensuring it meets the expected academic and scientific standards.
  - **Advisor**: Plays a crucial role in providing guidance, reviewing deliverables, and ensuring methodological rigour throughout the project.
- **Other Stakeholders**:
  - **Academic Community**: Seeks novel contributions and advancements in the comparison of database technologies.
  - **Data Regulatory Entities**: They may impact ethical and procedural considerations, for example, Environmental Protection Agencies (EPA), both regional and national entities may have an interest in the project's potential environmental implications, as energy efficiency directly relates to reducing carbon footprints.

2.2.1.2 Benefits, Constraints and Assumptions

The project delivers multiple benefits, addressing both technical and societal aspects. Some benefits were identified:

- Improved performance in Java applications through database optimisation.
- Reduced energy consumption in database operations.
- Easier choice between relational and non-relational databases.
- Significant academic contribution when comparing emerging technologies.
- Experience gained for future experiments and studies.
- Reduction in carbon emissions through more sustainable programming practices.
- Positive economic impact for companies that adopt more efficient practices.

The project is subject to several constraints that may impact planning and execution:

- Conditions that I don't control and that restrict my planning. These could be dates and costs.
- Limitations on the availability of real data for performance and energy tests.
- Ethical restrictions on the use of real or sensitive data during testing.
- Budget restrictions for the necessary infrastructure (e.g. servers, computing resources).
- Restrictions on access to production environments to carry out tests on real systems in companies or organisations.
- Complexity in comparing relational and non-relational databases, as each has its characteristics that are difficult to standardise.

The project relies on several key assumptions to guide its methodology and ensure valid results:

- Conditions decided on due to lack of information could give rise to risks.
- The choice of database significantly impacts energy and time consumption.
- The amount of data analysed impacts energy and time consumption.
- Tools and frameworks used are representative of the state of the art.
- Performance and energy consumption can be measured consistently and reliably.
- The studies chosen are sufficiently representative of the academic context.
- Simulated and real data have similar behaviours in the study context.
- Frameworks and technologies will continue to be relevant for years.
- The scientific community accepts the energy consumption and performance metrics adopted.
- Developers and software architects can easily adapt the study's conclusions.

2.2.2 Scope of Work

The research methodology aligns with the WBS phases and ensures a thorough exploration of performance-energy tradeoffs in Java applications. Figure 1 shows the elements schematically and Table 3 shows them describing its progress criteria.

WBS	Type	Description/Acceptance Criteria	Progress Criteria
1. Thesis	Project	Dissertation Project	Dissertation Preparation 20% Design 40% Implementation 60% Analysis of Results 85% Conclusion 100%
1.1 Dissertation Preparation	Phase	Includes all preparatory work required to establish a foundation for the dissertation.	Background 20% Literature Review 35% Project planning 85% Document Delivery 100%

1.1.1 Formalization	Deliverable	Definition of the thesis problem, its characteristics, objectives, and development process.	Draft completed 50% Formal approval 100%.
1.1.2 Background	Task	Analysis of historical and current information, leading to a comprehensive background report.	Research completed 50% Report finalised 100%.
1.1.3 Literature Review	Task	Conduct a detailed literature review to support project objectives, including research questions.	Selection of key sources - 50% Review completed 100%.
1.1.4 Skills Management	Task	Identification of required skills and strategies for its development.	Required skills 30% Assessment of current skills 50% Strategies for development 75% Impact of skills 100%
1.1.5 Project Charter	Task	Define the project scope, objectives, and stakeholders, documented in a formal project charter.	First draft 50% Charter completed and approved 100%.
1.1.6 WBS	Task	Develop the Work Breakdown Structure (WBS) to organise all phases and deliverables.	Preliminary structure 50% WBS approved 100%.
1.1.7 Plan	Task	Prepare a comprehensive project plan, identifying activities, dependencies, and deadlines.	Draft plan 50% Finalised and approved plan 100%.
1.1.8 Preparation Document Revision	Deliverable	Revise the preparation document to ensure all information is accurate and aligns with project goals.	Document to revision 50% Document approved 100%.
1.1.9 Preparation of Document Presentation	Deliverable	Present the preparation document for evaluation.	Presentation prepared 50% Completed presentation 100%.
1.2 Design	Phase	Encompasses activities to select tools, frameworks, and establish an experimental design.	Tools and frameworks selection 50% Experimental Design 100%
1.2.1 Tools and Frameworks Selection for Experiment (s)	Task	Evaluate and select appropriate tools and frameworks for the project, ensuring technical alignment with goals.	Tool selection 50% Frameworks selection 100%
1.2.2 Experimental Design	Task	Develop a detailed experimental design for hypothesis validation and project execution.	Draft design 50% Final design approved 100%.

1.3 Implementation	Phase	Includes setting up and testing the systems required for project execution.	Testing environment 50% Database configuration 100%
1.3.1 Testing Environmental Implementation	Task	Set up and validate the testing environment to ensure readiness for experiments.	Environment configured 50% Fully operational 100%.
1.3.2 Database Configuration	Task	Configure and test databases to ensure they meet the project's requirements.	Database setup 50% Fully functional and approved 100%.
1.4 Analysis of Results	Phase	Focused on analysing data obtained from experiments.	Performance analysis 25% Energy analysis 50% Hypothesis tests 75% Results comparison 100%
1.4.1 Performance Analysis	Task	Evaluate the system's performance using defined metrics and benchmarks.	Data collection 50% Final analysis approved 100%.
1.4.2 Energy Analysis	Task	Evaluate the system's energy efficiency using defined metrics and benchmarks.	Data collection 50% Final analysis approved 100%.
1.4.3 Hypothesis Tests	Task	Conduct hypothesis tests to validate the experimental results.	Initial tests 50% Finalised and validated results 100%.
1.4.4 Results Comparison	Task	Compare performance and energy results to obtain the tradeoffs.	Preliminary comparison 50% Finalised report 100%.
1.5 Conclusion	Phase	Encompasses the final steps of the project, including documentation, validation, and dissemination of findings.	Report 50% Report validation 75% Presentation 95% Article publication 100%
1.5.1 Report	Deliverable	Prepare a comprehensive report detailing the project's findings, methodology, and conclusions.	Initial draft 50% Validation report 100%.
1.5.2 Report Validation	Deliverable	Review and validate the final report with the advisor.	Report reviewed 50% Report approved 100%
1.5.3 Presentation	Deliverable	Develop and deliver a presentation.	Draft presentation 50% Completed and delivered 100%.
1.5.4 Article Publication	Deliverable	Prepare and submit an article for publication to share project findings with the wider academic or professional community.	Published article 100%.

Table 3 - WBS Dictionary

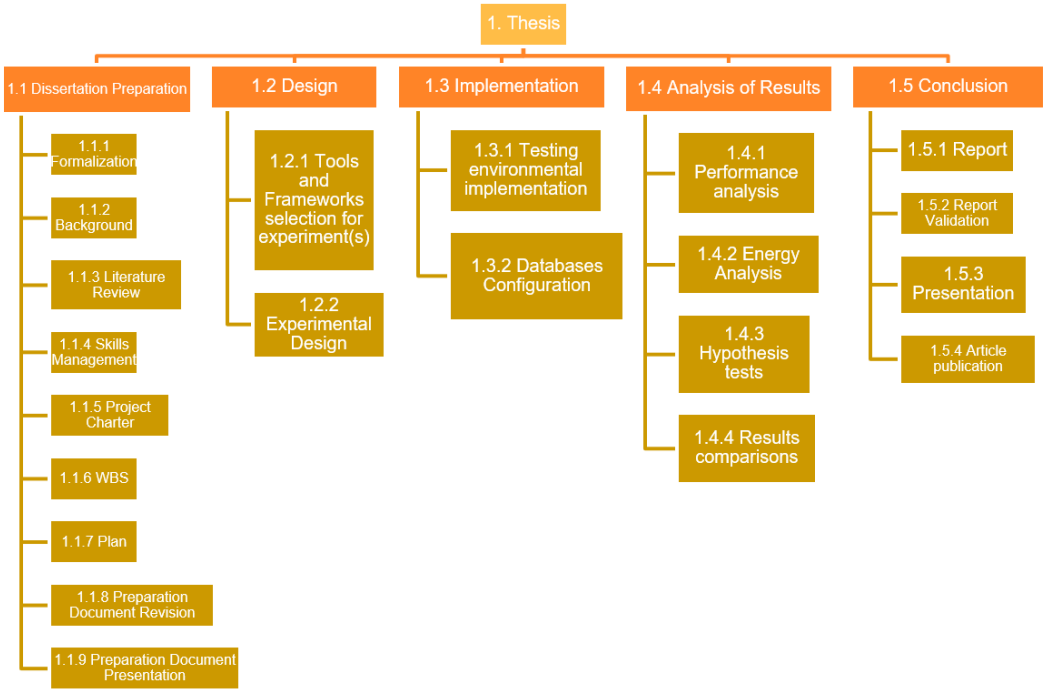


Figure 1 - WBS Project Deliveries

2.2.3 Integrated Project Schedule

The Gantt chart, with its Timeline, is available in Appendix A. It outlines major milestones and deliverables with estimated durations:

- **Schedule Overview:**
  - **Formalisation:** October 1, 2024 - October 16, 2024 (12 days).
  - **Background:** October 17, 2024 - November 6, 2024 (16 days).
  - **Literature Review:** November 7, 2024 - December 7, 2024 (22 days).
  - **Project Charter**: December 8, 2024 - December 10, 2024 (3 days).
  - **WBS**: December 11, 2024 - December 13, 2024 (3 days).
  - **Planning**: December 14, 2024 - December 15, 2024 (2 days).
  - **Preparation Document Revision**: December 18, 2024 - January 4, 2025 (14 days).
  - **Preparation Document Presentation**: January 1, 2025 - January 4, 2025 (4 days).
  - **Tools and Frameworks Selection**: February 17, 2025 - March 2, 2025 (11 days).
  - **Experimental Design**: March 3, 2025 - March 10, 2025 (6 days).
  - **Testing Environment Implementation**: March 11, 2025 - March 26, 2025 (12 days).
  - **Databases Configuration**: March 27, 2025 - April 17, 2025 (15 days).

- **Performance Analysis**: April 18, 2025 - April 24, 2025 (5 days).
- **Energy Analysis**: April 25, 2025 - April 30, 2025 (4 days).
- **Hypothesis Testing**: May 1, 2025 - May 8, 2025 (6 days).
- **Results Comparison**: May 9, 2025 - May 15, 2025 (5 days).
- **Report Writing**: May 16, 2025 - May 27, 2025 (10 days).
- **Report Validation**: May 28, 2025 - June 29, 2025 (33 days).
- **Final Presentation**: June 29, 2025 - June 29, 2025 (0 days).
- **Article Publication**: June 29, 2025 - June 29, 2025 (0 days).

To ensure the project's success, all tasks must be completed within the deadlines specified in the project schedule.

2.2.4 Costs

The project may incur various costs depending on the resources and tools used:

- Software licenses may be required depending on the tools chosen to carry out the tests (such as Hibernate for ORM, and JPA for persistence, among others). Tools such as IntelliJ IDEA may have associated costs if commercial versions are used. However, with the ISEP academic license, IDE's costs will be covered.
- If the project involves experiments that require dedicated servers or clusters to simulate large volumes of data, there will be infrastructure costs, either for cloud servers or the purchase of physical equipment. ISEP infrastructure provides remote servers that allow for the creation of databases covering these costs.
- The tools chosen to measure energy consumption and performance are open source, making some specialised equipment unnecessary.
- Submitting the paper to conferences or journals may incur registration or publication fees, such as fees for submitting papers to international conferences or costs related to publishing articles in scientific journals.
- The ISEP academic license provides access to various databases containing relevant documents and articles on the subject, thereby covering these types of costs. However, databases not included in the license will be excluded from consideration.
- Microsoft Office must be used, so the ISEP license covers the product's activation. However, it can be replaced with other tools like LaTeX.

From the previous costs enumerated, it is possible to conclude that the investigation has no monetary costs associated.

2.2.5 Risks Identification and Management

The project involves several potential risks that could impact its execution and outcomes. By identifying and analysing these risks, the project aims to establish mitigation strategies to ensure the reliability and success of the research. Those risks were identified and described in the Table 4.

Description	Motive	Effect
Uncontrolled variables or configurations that affect performance.	Lack of total control over the experimental environment.	Impact on the results and accuracy of the analysis due to variability in the data.
Frequent changes in the technologies used and the tool landscape.	Constant evolution of technology and frameworks.	The data collected may be irrelevant or out of date, affecting validity.
The choice of tools can lead to unrepresentative conclusions.	A bad analysis of the tool options.	The results may not be valid for other contexts.
The need for in-depth knowledge of specific tools can be an obstacle.	Technical requirements to understand specific technologies in depth.	Lack of experience can delay research and jeopardise the quality of experiments.
Too many tasks or a lack of planning can affect the progress of the project.	Time pressure and many responsibilities are on the author.	Delays in the timetable and a lack of quality can affect the delivery of the dissertation.

Table 4 - Project Risks Identification

For each risk, it was identified:

- Probability (1-5) - The author sourced a rough estimate of how likely this is to occur.
- Impact (1-5) - Rough estimate of how significant the impact of this risk is.
- PI Score - Probability multiplied by Impact.
- Expected Result, No Action - What will happen if the risk becomes an issue, and no action is taken?
- Risk Response Type - Decision made by the author on how to respond to this risk.
- Response description - How do you know it is time to put the response into play?

Therefore, the analysis done for each risk can be summarised as:

- **Uncontrolled Variables or Configurations That Affect Performance –** The probability of this risk occurring is moderate (3), with a low impact score of 2, resulting in a PI score of 6. If no action is taken, the accuracy and reliability of the analysis may be compromised due to variability in the data. This risk will be accepted as it is challenging to eliminate all external variables. Monitoring the setup rigorously will ensure any unexpected changes are documented and their effects mitigated.
- **Frequent Changes in the Technologies Used and the Tool Landscape –** This risk has a low probability of occurrence (1) and a low impact score of 2, giving a PI score of 2. Without any intervention, the collected data could become outdated, affecting its validity and relevance. The decision is to accept this risk due to its minimal impact, and actions such as continuous research into emerging technologies will help mitigate its effects if they arise.
- **Choice of Tools Can Lead to Unrepresentative Conclusions –** The probability of this risk materialising is moderate (3), with a moderate impact score of 3, resulting in a PI score of 9. The conclusions of the research may lack relevance or applicability in broader

contexts. To mitigate this, a thorough investigation into tool options and their applicability will be conducted during the project. The effectiveness of tools will be regularly reassessed to ensure their relevance to the research objectives.

- **The Need for In-Depth Knowledge of Specific Tools Can Be an Obstacle –** This risk carries a moderate probability of occurrence (3) and a high impact score of 4, yielding a PI score of 12. If unmanaged, this could delay the research and reduce the quality of experiments due to a lack of expertise in the chosen tools. To mitigate this, additional time and resources will be allocated for learning and mastering the tools. If the knowledge of the chosen tools is found insufficient to analyse the results, external expertise or training will be sought.
- **Too Many Tasks or Lack of Planning Can Affect the Progress of the Project –** This risk is assessed with a low probability of occurrence (2) but a moderate impact score of 3, resulting in a PI score of 6. If not addressed, the project timeline may be delayed, and the quality of deliverables could diminish. To mitigate this, careful task prioritisation and time management will be implemented. If the time allocated for completing tasks is insufficient, adjustments to the schedule and additional resources will be deployed to ensure project milestones are met.

3 Background

This chapter provides essential theoretical knowledge about key concepts of relational and non-relational databases, starting with some insights about them in the sections 3.1 and 3.2, followed by concepts of persistence layers in Java, such as Java Database Connectivity and Jakarta Persistence API, presented in 3.3. Also, it gives some insights into Performance and Energy attributes and available tools to measure them, available in sections 3.4 and 3.5.

3.1 Relational Databases

Relational databases organise data into tables consisting of rows and columns. This tabular structure facilitates efficient data management and retrieval. The relational model is grounded in mathematical theory, specifically set theory and relational theory, ensuring a robust framework for data operations.

Relational databases follow the principle of ACID, which stands for Atomicity, Consistency, Isolation, and Durability [14], [15].

Atomicity ensures that a transaction is treated as a single, indivisible unit. This means that all operations within a transaction must be completed; if any part fails, the entire transaction is rolled back, leaving the database unchanged. For example, in a bank transfer, the debit from one account and the credit to another must both succeed or fail.

Consistency guarantees that a transaction takes the database from one valid state to another, maintaining its integrity. After completing the transaction, all data must conform to the defined rules and constraints, ensuring data validity. For example, a transaction will not be committed if it violates data integrity rules (e.g., foreign key constraints).

Isolation ensures that concurrent transactions do not affect each other's outcomes. Transactions are executed in isolation, meaning partial updates from one transaction are not visible to others until the transaction is complete. This prevents issues such as dirty, non-repeatable, or phantom reads.

Durability ensures that once a transaction is committed, it remains permanent in the database, even in the event of a system failure. The changes are recorded in non-volatile memory, such as a hard disk, so the data won't be lost.

Relational databases are characterised by their structured schema [15], which forms the backbone of their operation.

3.2 Non-Relational Databases

Non-relational databases, also known as NoSQL databases, which stand for “Not only SQL”, have seven important characteristics [2]:

- are not based on a relational approach,
- scale horizontally,
- are often open-source products,
- don’t need a defined schema, providing high flexibility,
- provide an API for the integration of other software products,
- use a decentralised architecture for the easy replication of data,
- follow the BASE principle (Basically Available, Soft State, Eventually Consistent).

The BASE principle consists of [6] and [14]:

- **Basically Available** - The system should remain responsive and continue handling requests even if some servers suffer from failures.
- **Soft State** - The state of the database can change over time, even without active writing operations, due to ongoing synchronisation and background processes.
- **Eventual Consistency** - When data is written to one server, updates must be distributed to other servers; during this process, the data might be temporarily inconsistent, but it will eventually reach a consistent state across all nodes.

Non-relational databases encompass various types, including key-value stores, column stores, and document stores, each tailored to specific data storage and retrieval needs [14].

3.2.1 Key Value Stored Databases

Key-value stores, often called dictionaries or hashes, have gained popularity in NoSQL systems due to their ease of scalability and rapid, seamless growth. The fundamental idea is that a globally distributed hash table contains keys that map to database servers spread across the globe. Each data item is assigned a unique key generated through a specific formula, which is then stored in a lookup table or directory [14]. When data retrieval is required, the key is used to locate and fetch the data [14]. Also, buckets serve as logical groupings of keys, though they do not physically group data. Each bucket can contain duplicate keys, requiring both the bucket and key to retrieve a value [1].

The read and write operations of key-value databases are Get, Put, Multi-get and Delete:

- Get(key) – returns the value associated with the provided key.
- Put(key, value) – associates the value with the key.
- Multi-get(key1, key2,..., keyn) – returns the values associated with the list of keys.
- Delete(key) – removes the entry for the key from the data store.

Memcached [16] and Redis [17] are two popular examples of key-value databases. Both are in-memory data stores, which refers to data that is stored directly in a computer's RAM rather than on disk storage, that use a key-value pair model for storing and retrieving data.

3.2.2 Column-Store Databases

Column-oriented databases are often viewed as a hybrid between NoSQL and traditional relational databases. While they offer a row-and-column structure, they do not adhere to the strict schema rules of relational databases, allowing for greater flexibility. Unlike row-oriented databases, column-oriented databases store and process data by columns rather than by rows. It also offers Column families that group related columns and allow for an unlimited number of columns to be added at runtime [1].

These databases store data so all the values of a specific column are kept together, enabling highly efficient data retrieval and processing. This design originated in the realm of analytics and business intelligence, making columnar databases ideal for high-performance applications [14]. For example, retrieving the prices from a database of a million products would be time-consuming in a traditional relational database, as it would need to scan through each row to extract the price from each record. In contrast, a column-oriented database can access the entire "price" column in a single disk read, making the retrieval process much faster and more efficient.

One major advantage is that columnar databases only perform I/O on the blocks related to the columns being accessed. This means that, unlike row-based systems where entire rows (including unused columns) are read, columnar databases minimise I/O overhead by focusing only on the relevant columns. Also, by storing similar data types together (e.g., dates, text), memory utilisation is more efficient, further enhancing query performance. Additionally, columns can be highly compressed, reducing storage requirements and speeding up query processing [14].

Some examples of Column databases are BigTable [18] and Apache Cassandra [19].

3.2.3 Document Stored Databases

Document-oriented databases store data as documents, with groups of these documents organised into collections. A collection can contain any number of documents, each of which can be of a different type, making these databases schema-less. This flexibility allows for the easy storage and management of various types of data, facilitating rapid development and adaptation to changing requirements.

Unlike traditional relational databases that store data in rows and columns, document databases optimise data storage and access for documents. Each document represents a record with multiple fields that can hold different types of data, including nested structures and arrays. Most modern document databases use JSON (JavaScript Object Notation) format for documents, providing a format that is both human-readable and well-integrated with web technologies and modern programming languages [14].

The following examples, in Code Snippet 1, show the documents that represent the same type of information in a system, which are products; these products have different properties associated:

```
[
   {
      "productName":"Wireless Headphones",
      "details":{
         "Brand":"SoundMax",
         "Model":"X300",
         "Price":"$59.99"
      }
   },
   {
      "productName":"Gaming Keyboard",
      "details":{
         "Brand":"KeyPlus",
         "Backlight":"RGB",
         "Connection":"Wired",
         "Price":"$39.99"
      }
   },
   {
      "productName":"Smartwatch",
      "details":{
         "Latitude":"37.7749",
         "Longitude":"-122.4194",
         "BatteryLife":"24 hours"
      }
   }
]
```

Code Snippet 1 – Document structure in a Document Stored Database

Document databases support horizontal scalability, allowing data to be distributed across multiple servers to handle large amounts of data while maintaining high availability. This makes them well-suited for applications that require fast read and write operations, as well as complex, varied data structures.

One example of a popular document database is MongoDB [20], known for its flexible, JSON-like structure, and Couchbase [21]. Also, Elasticsearch is considered a document database [22].

3.3 Persistence Layer

In the context of the layered architecture pattern, the persistence layer serves as a critical intermediary that facilitates the structured management of data interactions within software systems. Figure 2 presents an architecture showing the layers that a generic application has, making it possible to see where the Persistence Layers fit.

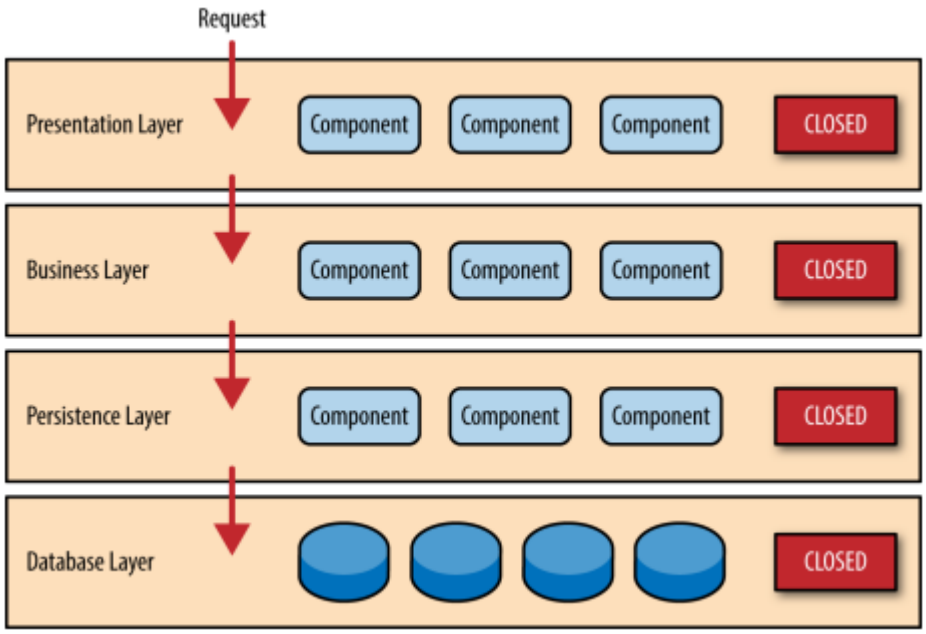


Figure 2 - Architecture Layers Structure

Source: [23]

Positioned between the business layer and the database layer, it encapsulates the functionality required for data operations such as retrieval, insertion, updating, and deletion, it provides a consistent and unified interface for higher-level application components. This abstraction not only decouples the business logic from the intricacies of database operations but also promotes a modular architectural design, enabling clear separation of responsibilities and fostering maintainability in complex systems.

There are examples of development tasks, like migration from one type of persistence layer implementation to another, that are not easy to perform, due to the requirement to rewrite all the queries and rethink the structure of the database and backend models [24]. These difficulties allow us to conclude that it is essential to think and decide how the persistence layer of an application should be built.

Jakarta Persistence APIs (JPA) and Java Database Connectivity (JDBC) are examples of persistence layer technologies that have a crucial role in data management and migration processes. These tools exemplify the diversity in persistence layer implementations, maintaining various application requirements and being fundamental during system migration efforts.

3.3.1 Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a powerful API that enables seamless interaction between Java applications and various database systems. Designed for flexibility and efficiency, JDBC allows developers to use a single, standardised interface to interact with databases, regardless of the specific database management system (DBMS) involved, as long as the appropriate JDBC-compliant driver is available. Developers can write the queries themselves to send or retrieve the necessary data from the databases. This implies that the developer does all the work connecting to the database manually.

The queries and procedures are written using Structured Query Language, SQL. Each database has its own SQL statement syntax. Therefore, software developers should write them in the specific database syntax in which the data is stored.

One of its key strengths is its ability to work with both local and remote data sources, supporting advanced SQL-92 compliant drivers while also being compatible with older legacy systems. This flexibility ensures that applications can adapt to a wide range of database environments without significant changes to the underlying code [25].

JDBC also provides robust support for complex data types, such as binary large objects (BLOBs). This feature makes it possible to store and retrieve multimedia data, including images and videos, directly from databases, significantly enhancing its utility in diverse application scenarios [25].

Another standout feature of JDBC is its modular design, which ensures that switching between different database drivers or even databases can be done without rewriting the core application logic. This modularity not only simplifies development but also promotes scalability and maintainability [25].

The JDBC API provides transaction features. A transaction is a group of one or more statements that are executed as a single unit, ensuring that either all the statements are completed or none are carried out [26]. So, for example, if the second statement fails, the first one does not impact

the database. They are critical for ensuring data integrity and consistency in a database following ACID properties.

In JDBC, transactions are typically controlled through the *Connection* object. By default, JDBC connections operate in auto-commit mode, where each SQL statement is treated as a transaction and is committed immediately after execution. To manage transactions manually, auto-commit must be disabled using the *connection.setAutoCommit(false)*. This allows multiple SQL statements to be grouped into a single transaction [25]. After executing the desired operations, the transaction can either be:

- Committed using *connection.commit()* to make all changes permanent, or
- Rolled back using *connection.rollback()* to undo all changes made during the transaction.

Proper transaction management is essential to handle errors and maintain database reliability, especially in applications involving multiple interdependent operations.

On Code Snippet 2, there is an example of a JDBC connection.

```
String query = "SELECT email FROM customer_table "
        + "WHERE unique_id = ?";
ResultSet rs = jdbcCon.prepareStatement(query)
        .setInt(1, id).executeQuery();
String email = rs.next() ? rs.getString("email") : "";
```

Code Snippet 2 - Example of JDBC

Source: [5]

Considering all the heavy work that JDBC requires, the Jakarta Persistence API (JPA) has been developed.

3.3.2 Jakarta Persistence API (JPA)

Jakarta Persistence API is a JAVA standard that maps data stored in relational databases with Plain Old Java Objects (POJO) [5]. The method that connects relational databases and Java applications is called Object-Relational Mapping, or ORM.

JPA provides the *EntityManagerFactory* interface, which allocates and releases resources. After finishing the mapping process, the resources are released, and the factory is closed. To initiate the persistence process, an *EntityManagerFactory* object is called by an instance of the *EntityManager* interface. The *EntityManager* works within a persistence context to handle CRUD (create, read, update, delete) operations on entities. In JPA persistence, SQL operations are managed via transaction and query objects, which handle Data Definition Language (DDL) operations [27].

JPA performs the data modelling with annotations such as @annotations name or XML tags. Due to the complexity of XML tags, it is not preferred for programming strategies, as the

annotations are much simpler. For example, imagine the need to add a new column to a table. Doing it with JDBC, it is required to do it manually in the database, then it is necessary to add the column in the query and then create the setter to retrieve it to the object in the application. With ORMs, for example, Hibernate, it is only necessary to write the property in the Java application with the required annotations, and the framework will take care of the rest.

The four primary strategies for connecting an ORM to a database are Native Access, JPQL API Access, Criteria API, and Managed Entity, each offering unique approaches to interact with and manage database operations [5].

The **Native Access Strategy** refers to the use of raw SQL queries in Java applications through the Java Persistence API (JPA). In this approach, developers leverage JPA's API to execute native SQL commands directly, bypassing some of the abstractions typically provided by JPA. These raw SQL queries are referred to as "native queries", because they connect directly with the database's native query language [5].

As illustrated in Code Snippet 3, a native query is executed using the *EntityManager* provided by JPA. The process involves creating a named query (e.g., *Customer.getEmailSQL*) and setting any required parameters before retrieving the results. This allows developers to gain greater control over the database interaction while still benefiting from JPA's infrastructure for parameter binding and result handling.

```
EntityManager em = // ...
String email = (String)
        em.createNamedQuery("Customer.getEmailSQL")
                .setParameter(1,id).getSingleResult();
```

Code Snippet 3 - Native Access Strategy example

Source: [5]

Unlike plain JDBC, where manual mapping between result sets and Java objects is necessary, the Native Access Strategy with JPA encourages declaring the mapping logic using configurations or annotations within the framework. This simplifies development and reduces the likelihood of errors while still allowing the use of raw SQL for complex or performance-critical operations.

The **JPQL API Access Strategy** is a data access method that leverages the Jakarta Persistence Query Language (JPQL), a high-level query language with SQL-like syntax designed specifically for interacting with databases through Java objects. JPQL acts as an abstraction layer over raw SQL queries, enabling developers to define queries independently of the underlying database's SQL dialect. Developers interact with Java class names and fields, which are mapped to corresponding database tables and columns rather than directly referencing database-specific identifiers. This abstraction simplifies the query creation process and enhances the portability of the code, as JPQL queries are database-independent and do not require modifications to adapt to different database platforms [5].

As demonstrated in Code Snippet 4, a query is executed using the *EntityManager* to create a named JPQL query (e.g., *Customer.getEmailJPQL*). Parameters can be set using the *setParameter* method, and results are retrieved in a type-safe manner. This method ensures that queries align with the object-relational mappings defined in the JPA configuration, improving code readability and maintainability.

```
EntityManager em = // ...
String email = em.createNamedQuery(
        "Customer.getEmailJPQL", String.class
        ).setParameter(1,id).getSingleResult();
```

Code Snippet 4 - JPQL Strategy example

Source: [5]

The **Criteria API Strategy** is a query-building approach provided by the Java Persistence API that enables developers to construct queries programmatically using Java objects and method calls. Unlike raw SQL or JPQL, which involve writing queries as plain text, the Criteria API allows for dynamic and type-safe query creation entirely within the Java programming language. This approach relies on the *CriteriaBuilder* class, which serves as a factory for creating different components of a query. Developers use the API to define the query structure step-by-step, such as specifying the target entity (*Root*), parameters (*ParameterExpression*), conditions (*Predicate*), and result selection. Each part of the query is built using methods provided by the Criteria API, ensuring that queries are type-safe and checked at compile time.

Code Snippet 5 demonstrates how to construct a query using the Criteria API. First, a *CriteriaBuilder* instance is obtained from the *EntityManager*. Then, a *CriteriaQuery* object is created, specifying the return type (*String*). The Root defines the target entity (*Customer*), and a *Predicate* is created to establish a condition. Finally, the query selects the desired field (email) and applies the condition, with parameters being set dynamically before execution.

```
EntityManager em = // ...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<String> query = cb.createQuery(String.class);
Root<Customer> root = query.from(Customer.class);
ParameterExpression<Integer> idParameter =
        cb.parameter(Integer.class);
Predicate predicate = cb.equal(idParameter,
root.get(Customer_.id));
String email = em.createQuery(query
                .select(root.get(Customer_.email))
                .where(predicate)
        ).setParameter(idParameter, idValue)
        .getSingleResult();
```

Code Snippet 5 - Criteria API Strategy example

Source: [5]

The Criteria API Strategy is particularly advantageous when queries need to be generated dynamically at runtime, as it allows for flexible query construction without relying on string

concatenation or manual parsing. Additionally, its programmatic nature ensures compatibility across different database platforms, as the JPA provider translates the Criteria API queries into the appropriate SQL for the underlying database.

The **Managed Entity Strategy** involves working with entities that are mapped to database records and actively managed by the Object-Relational Mapping (ORM) framework, such as JPA. These managed entities are Java objects that represent rows in a database table, and their state is synchronised with the database by the ORM. This means any changes made to these objects are automatically monitored and persisted in the database without requiring explicit SQL or query execution [5].

In Code Snippet 6, the *EntityManager* is used to find a *Customer* entity based on its class and identifier (*id*). The email field is then accessed directly through the getter method. This approach abstracts away the complexities of query writing and focuses on interacting with objects rather than database tables.

```
EntityManager em = // ...
Customer c = em.find(Customer.class,id);
String email = c.getEmail();
```

Code Snippet 6 - Entity Access Strategy example

Source: [5]

3.3.3 Examples of existing JPA Technologies

Hibernate and EclipseLink are two Object-Relational Mapping frameworks utilised in modern application development, being among the most widely adopted [27].

3.3.3.1 Hibernate

Hibernate offers a Hibernate Query Language, HQL, which is an advanced query language that allows the execution of CRUD operations from Java classes into relational databases. This means that any query the developer writes in HQL will be transferred to the database as a specific query.

Similarly to the JPA process, a configuration object serves as the *SessionFactoryManager*, which initiates the persistence process and establishes the link between Java entities and the corresponding database tables. The *SessionFactoryManager* is called by the session instance to carry out SQL operations. During this process, the session instance manages caching and requires the explicit use of the *Close()* method to prevent memory leaks. For database transactions, a session object in the Hibernate API invokes a transaction object, which operates within the session to perform tasks via the transaction manager and underlying queries. The *SessionFactoryManager* uses the JPA entity, *EntityFactoryManager* [27].

Figure 3 illustrates very well the similarity between the Hibernate process and the JPA process.

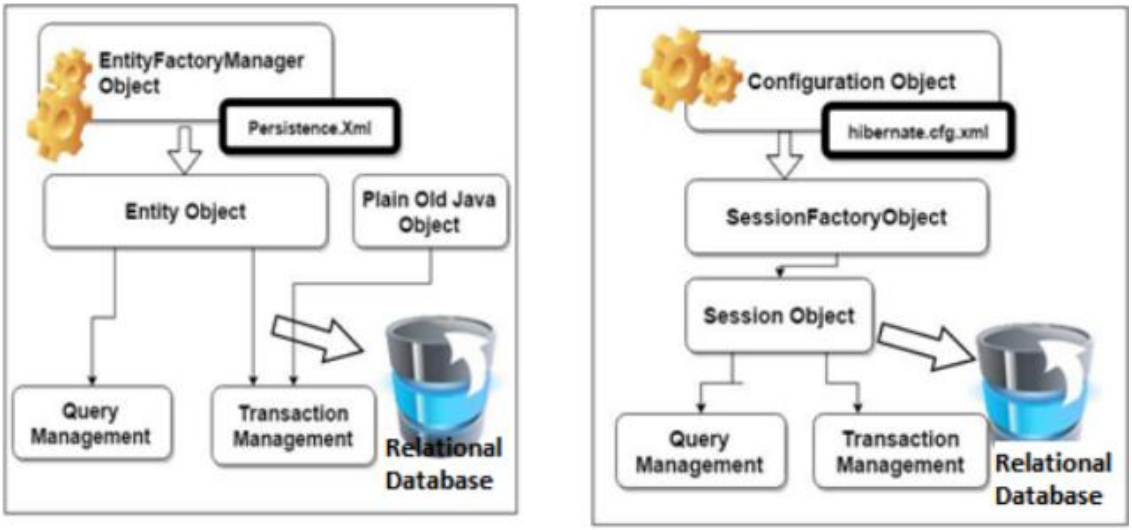


Figure 3 - Hibernate JPA High-Level Architecture

Source: [27]

However, this framework increases the complexity of the solution [24]. A team leader nearly lost his job due to the complexity of how Hibernate was connected to its application's persistence layer [28].

3.3.3.2 EclipseLink

EclipseLink is a sophisticated JPA provider originally developed as Oracle's TopLink and later made open-source. It simplifies complex database interactions and enhances application performance by providing persistence services. EclipseLink uses an efficient ORM framework to connect Java applications with relational, non-relational, and cloud databases [27].

EclipseLink operates through a cohesive architecture, described on Figure 4, that blends configuration, metadata, and runtime enhancements to bridge the gap between Java objects and database schemas. The *EntityManagerFactory* and *EntityManager* play integral roles in orchestrating the persistence process, while project classes, descriptors, and XML metadata define the mappings and transformations. With its robust runtime mechanisms and advanced functionalities, EclipseLink JPA is well-suited for building scalable and efficient database-driven applications [27].

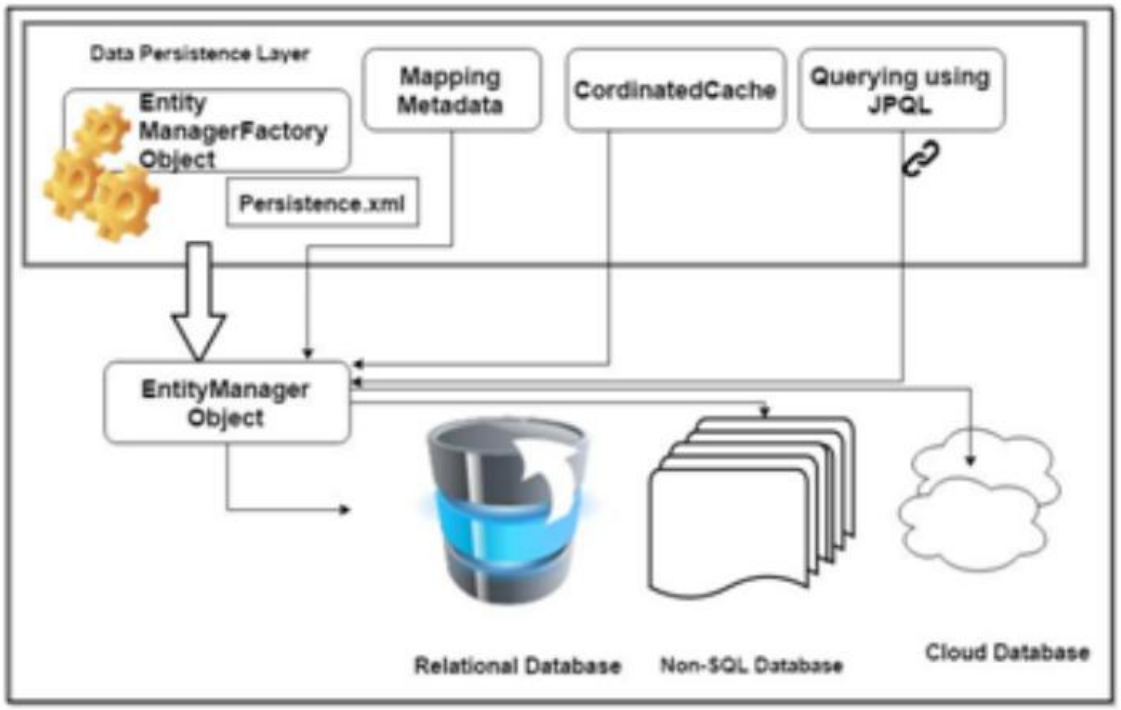


Figure 4 - EclipseLink Architecture

Source: [27]

3.4 Performance

Performance testing is conducted to evaluate a system's behaviour under specified performance requirements. It ensures the system meets user expectations concerning speed, accuracy, and resource utilisation, which are critical, particularly in real-time systems. Performance testing identifies issues like latency, throughput, and resource consumption that directly influence user satisfaction and system acceptance. The lack of proper performance considerations has resulted in project risks and market failures. Performance requirements, typically non-functional, should be quantitatively specified to be measurable and testable. Performance testing tools employ controlled environments with simulated workloads to replicate user behaviour, enabling the identification of bottlenecks and compliance with specified requirements [29].

Performance testing focuses not only on system capabilities but also on the tester’s role. Metrics such as "number of bugs found" and "severity of bugs" are traditional, yet there is criticism of over-reliance on quantitative metrics. Insights from industry professionals reveal the importance of rigorous planning and execution in performance testing [30].

3.4.1 Performance metrics

Performance testing metrics help quantify the efficiency and quality of the testing process. The systematic evaluation using such metrics enables organisations to monitor and improve testing processes. Analysing these metrics allow conclusions about the productivity of scripting, execution, and defect identification during performance testing [29], [30], [31]:

- Response time
- Latency – The time delay between request initiation and completion.
- Throughput – The Amount of work processed within a given timeframe.
- Resource Consumption – Memory or disk space utilised during operations.
- Number of Bugs Found
- Severity of Bugs
- Quality of Bug Reports
- Rigorousness of Planning and Execution
- Ability for Bug Advocacy

3.4.2 Examples of existing tools

JMeter and Grafana k6 are two existing tools to measure performance, and they were chosen due to successful results from previous academic studies. These tools provide comprehensive features that enable users to simulate various scenarios and workloads, offering valuable insights into system behaviour under different conditions. Their functionality extends to evaluating key performance metrics, ensuring reliability and scalability in diverse environments [32].

3.4.2.1 JMeter

Apache JMeter is an open-source tool developed in Java by the Apache Foundation, initially released in 1998. It has gained popularity for offering functionalities comparable to proprietary load-testing tools. JMeter is primarily used for load testing and simulating traffic at the API or protocol level [32].

JMeter offers a comprehensive suite of features that facilitate effective performance testing [33]:

- **Protocol Support**: JMeter can test performance on static and dynamic resources and supports various protocols, including HTTP, HTTPS, SOAP, REST web services, FTP, JDBC, LDAP, and JMS.
- **Test Plan Creation**: Users can create detailed test plans that define the number of virtual users (threads), the ramp-up period, loop counts, and the specific requests. This allows the simulation of various load scenarios to assess system performance under different conditions.

- **Extensibility**: JMeter's architecture is highly extensible, allowing users to enhance its functionality with plugins and integrate it with other tools to meet specific testing requirements.

JMeter operates by simulating multiple users sending requests to a target server and generating a load to test the server's performance. The process involves several key components [33]:

- **Thread Groups**: Define the number of virtual users and the schedule for their requests. Each thread simulates a user, and the thread group configuration determines how users are introduced to the system under test.
- **Samplers**: Represent the actual requests sent to the server. JMeter provides various samplers for different protocols, such as HTTP Request samplers for web applications.
- **Listeners**: Collect and present test results. They provide visualisations and reports that help analyse the system's performance metrics under test.
- **Timers, Assertions, and Configuration Elements**: Timers introduce delays between requests to simulate real user behaviour; assertions validate responses to ensure correctness; and configuration elements set default values and variables for use throughout the test plan.

It also provides various listeners and reports to analyse the performance of the system under test. These tools help identify bottlenecks and measure response times, throughput, error rates, and other critical performance indicators [33].

3.4.2.2 Grafana k6

Grafana k6 is an open-source tool developed in 2017 by Grafana Labs, designed for performance testing. Written in Go, k6 is lightweight, efficient, and focused on load and stress testing. Unlike some older tools, k6 does not come with a built-in graphical user interface (GUI), but it allows integration with a GUI when using its cloud service [32].

Below is an enumeration of k6's core features and capabilities, illustrating how it empowers developers and quality assurance professionals to simulate real-world scenarios, analyse system performance, and streamline testing processes [34]:

- **JavaScript Scripting**: k6 uses JavaScript for writing test scripts, allowing developers to build realistic load tests using a familiar language. This approach supports the reuse of modules and libraries, enhancing the maintainability of test suites.
- **Command-Line Interface (CLI)**: k6 offers an intuitive CLI that simplifies test execution, making it accessible for modern engineering teams.
- **Extensibility**: k6 can be improved with additional functionalities, such as browser automation, support for various protocols, and integration with different result storage backends like Prometheus and Grafana.
- **Performance Testing**: It supports various types of performance testing, including stress, spike, and soak tests, enabling teams to evaluate system behaviour under different load conditions.

- **CI/CD Integration**: Designed with automation in mind, k6 integrates seamlessly into Continuous Integration and Continuous Deployment (CI/CD) pipelines, allowing teams to include performance testing of their development process.

K6 operates by running test scripts written in JavaScript, which define the behaviour of virtual users interacting with the system under test. The process involves several key components [34]:

- **Test Scripts**: Users write test scripts in JavaScript, specifying virtual users' actions, such as sending HTTP requests to specific endpoints. These scripts can include logic for setting test parameters, defining thresholds, and incorporating custom metrics.
- **Virtual Users**: k6 simulates multiple virtual users concurrently executing the test script, generating load on the target system. The number of users and the test duration can be configured to match specific testing scenarios.
- **Metrics and Thresholds**: During test execution, k6 collects various metrics, such as response times, request rates, and error rates. Users can define thresholds to set performance goals; if these are not met, it will indicate a failed test, providing immediate feedback on system performance.
- **Result Output**: k6 provides detailed reports of the collected metrics after the test. These results can be output to various backends, including time-series databases and visualisation tools like Grafana, facilitating in-depth system performance analysis.

3.5 Energy consumption

Energy consumption in software systems has become a critical issue with the growing reliance on portable, embedded, high-performance computers. Efficient use of energy is essential to delivering longer battery life in mobile devices, reducing operational costs in large-scale systems, and meeting sustainability goals. Software directly influences energy consumption by utilising hardware resources, like processors, memory, and storage. By understanding and managing these interactions, developers can design software that meets functional requirements and minimises power usage, ensuring better performance, longer device lifespans, and lower environmental impact [35], [36], [37].

Both hardware and software factors influence energy consumption in software systems. Power usage can be forecasted from hardware counters and operating system parameters, such as CPU usage and memory bandwidth. Statistical models provide a simplified explanation of complex interactions, enabling the identification of energy-intensive behaviours. These models are valuable for optimising power usage in high-performance and embedded systems where energy efficiency is critical [35].

In embedded applications, energy consumption optimisation focuses on runtime qualities such as memory utilisation and CPU load. Techniques like loop interchange and variable elimination minimise unnecessary memory access and improve computational efficiency. Profiling tools play a critical role in identifying energy-intensive code regions, enabling targeted optimisations while considering tradeoffs with maintainability [37].

3.5.1 Energy metrics

Energy consumption metrics provide insights across the software lifecycle to support sustainable development approaches. Non-intrusive monitoring techniques allow measurement in real time, which could be used to stimulate iterative improvement. A critical examination of green metrics identifies significant avenues of critical mismatches in standardisation and practice of energy conserving measures, i.e., there is a need for tailormade metrics in mobile, cloud, and embedded systems [36].

Software energy metrics bridge the gap between hardware performance and software design. Processor power consumption, driven by instruction execution and memory access patterns, forms the basis of most energy models. Hierarchical flowgraph-based evaluations enable comparisons of software implementations independent of specific hardware. This approach promotes energy-aware programming practices and better integration of energy efficiency as a software design parameter [38].

Some energy metrics were identified [35], [36], [37], [38]:

- Energy Consumption
- CPU Utilisation
- Memory Bandwidth
- Instruction Counts from Performance Counters
- Memory Access Frequency
- Runtime Resource Utilisation

3.5.2 Examples of existing tools

This section presents two existing tools to measure energy consumption. The author of this document received good feedback from Kepler, and JoularJx seems to be easy to implement and has good feedback on existing research.

3.5.2.1 Kepler

One existing tool to measure energy consumption is the Kepler project. It stands for "Kubernetes-based Efficient Power Level Exporter" and provides white-box energy monitoring capabilities [39].

Kepler is a tool designed to monitor and estimate the energy consumption of Kubernetes' workloads. It utilises eBPF (extended Berkeley Packet Filter) to collect performance metrics from the Linux kernel, such as CPU performance counters and tracepoints. These metrics and data from system filesystems like *sysfs* are processed using machine learning models to estimate the energy usage of individual pods and containers [40]. Its architecture is extensible, allowing the integration of various power models to accommodate diverse system architectures. It collects real-time power consumption data from node components through APIs like Intel's Running Average Power Limit (RAPL) for CPU and DRAM power, NVIDIA Management Library

(NVML) for GPU power, and Advanced Configuration and Power Interface (ACPI) for platform power [41]. Kepler employs regression-based trained power models to estimate energy consumption in environments lacking real-time power metrics [41].

The collected data enables Kepler to calculate the energy consumed by each process by proportionally dividing the power a resource uses based on the process's resource utilisation relative to the system's total utilisation. This process allows Kepler to aggregate power consumption metrics at the container and Kubernetes pod levels, providing detailed insights into the energy usage of cloud applications [41].

Kepler exports these energy consumption metrics in a Prometheus-friendly format, facilitating integration with monitoring and observability tools. This capability empowers users to monitor their containers' energy consumption, aiding in making informed decisions to achieve energy conservation goals [40].

3.5.2.2 JoularJx

Another tool to measure energy consumption in software systems is JoularJX, designed to run in Linux systems. JoularJX is a Java-based agent for power monitoring at the source code level. It is derived from the PowerJoular framework and utilises the Intel processor's Running Average Power Limit (RAPL) interface, accessed via the Linux Power Capping Framework, to take real-time measurements of system component energy consumption [42], [43].

At runtime, JoularJX initialises PowerJoular with appropriate parameters and monitors the Java process's power usage by tracking its PID, Process ID. It continues to collect CPU usage data on a one-second basis and allocates power usage to Java threads proportionally. At the same time, it monitors the stack trace of each thread every 10 milliseconds, sees how the method currently running is being executed and allocates energy usage statistically to the method. This allows one to produce exhaustive reports on all the methods invoked under the program, application code, and Java library methods. Or, power monitoring can be restricted to specific methods or packages to facilitate targeted analysis [43].

4 Literature review

This chapter presents the literature review, starting with the preparation done in the section 4.1, followed by the data collection process executed and its results in 4.2 and 4.3, ending with a discussion of the results and their conclusions in sections 4.4 and 4.5.

4.1 Preparation

Taking into consideration the problem description and objectives mentioned above in the sections 1.2 and 1.3, respectively, three research questions were used in this study:

- **RQ1**: How does the choice between relational and non-relational databases impact Java-based applications' energy consumption and performance?
- **RQ2**: How does query optimisation for relational databases affect energy consumption and performance compared to non-relational databases?
- **RQ3**: How does data volume influence query performance and energy consumption in relational and non-relational databases?

For the literature review, three key data sources were used to identify relevant studies, each renowned for its extensive collection of high-quality, peer-reviewed articles highly pertinent to information technology. These sources include the ACM Digital Library, b-on and IEEE Xplore, which collectively provided access to a diverse array of publications such as conference proceedings, journal articles, and online books.

Performance, Energy, databases and Java are the most relevant keywords.

The Code Snippet 7 was written as a search query based on the required keywords to filter documents from the data sources. This search query combines some search terms with **OR** and **AND** operators.

```
(
        (“performance” OR “energy”) AND
        (“databases” OR “SQL” OR “NoSQL”) AND
        “Java”
)
OR
(
        (“query optimisation” OR “data volume”) AND
        “performance” AND
        “energy” AND
        (“databases” OR “SQL” OR “NoSQL”)
)
```

Code Snippet 7 - Search query

Regarding the inclusion and exclusion criteria, they were defined as:

- **Inclusion Criteria**
  - **IC1**. The source explores the performance or energy efficiency of relational or non-relational databases.
- **Exclusion Criteria**
  - **EC1**. The source does not compare performance or energy efficiency on relational or non-relational databases.
  - **EC2**. The document is not written in English or Portuguese.
  - **EC3**. Exclude documents published before 2010.
  - **EC4**. Exclude all documents that require payment for their content.

4.2 Data collection process

In this study, a systematic data collection process was conducted following the PRISMA guidelines [44], as described in Figure 5. The methodology Trail-and-Error [45] was applied to develop the query mentioned in the section 4.1. It consists of trying out various combinations of keywords, checking if they produce meaningful and relevant literature, and refining the queries based on the outcome.

The query was divided into two blocks: the first block included the core search terms, while the second block consisted of additional keywords to refine the search further, especially about energy consumption. This structure allowed for more precise filtering and relevance in search results within these databases. It was applied in filtering abstracts, titles and keywords.

In the Identification phase, duplicate entries were eliminated to clean the dataset. During Screening, titles and abstracts were reviewed to apply the inclusion and exclusion criteria and narrow down the list to studies that seemed relevant. After that, the Retrieval step involved checking whether the full texts of these studies were accessible. Finally, in the detailed analysis phase, the available full-text documents were thoroughly evaluated to confirm their relevance to the research goals, ensuring that only high-quality and well-aligned studies were selected for the final review.

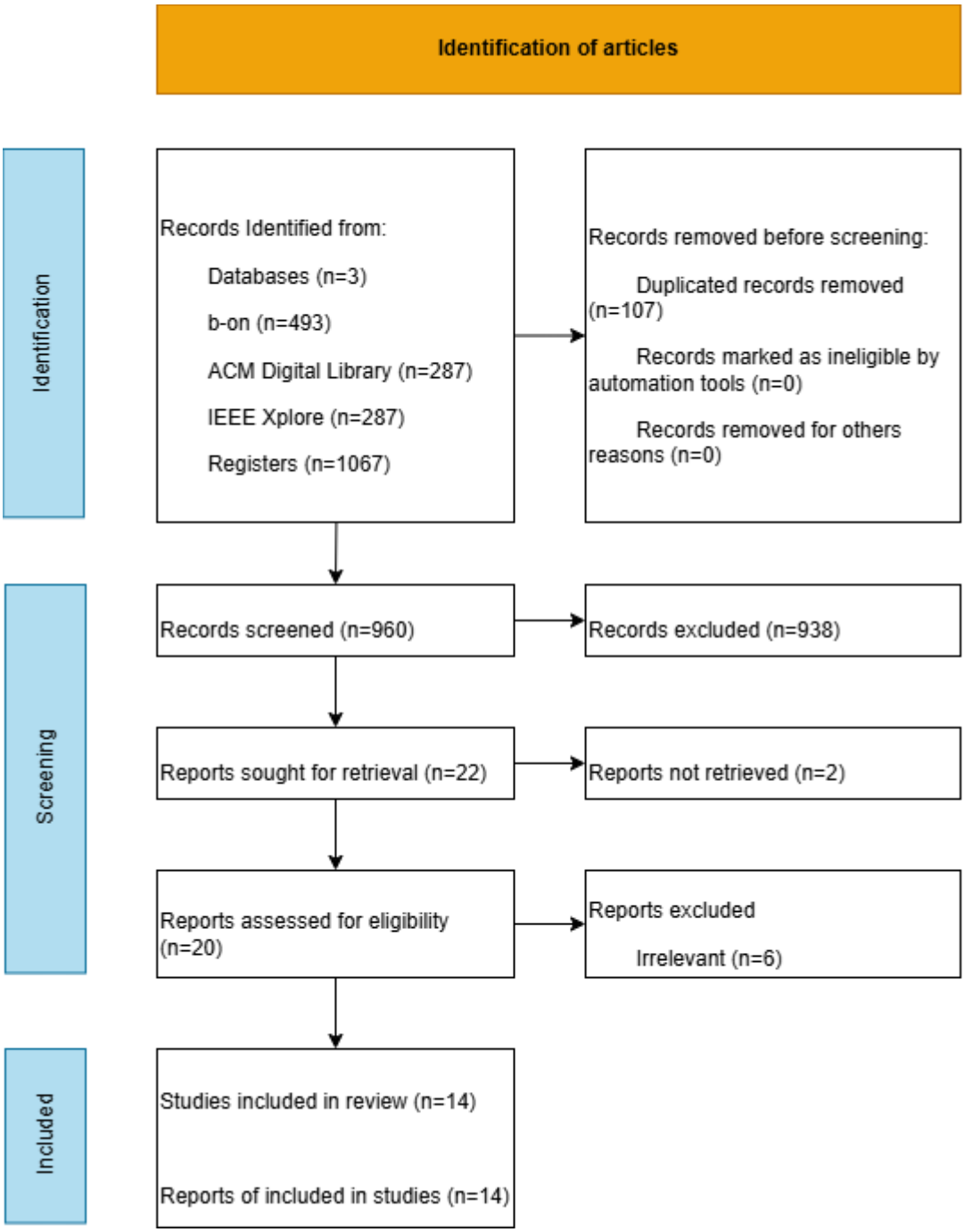


Figure 5 - Data Collection Process

4.3 Results

In Table 5, there is a correspondence between the studies and the research questions mentioned.

RESEARCH QUESTIONS	STUDIES
RQ1	[6], [46], [47], [48], [49], [50], [51], [52], [53]

RQ2	[54], [55]
RQ3	[56], [57], [58]

Table 5 - Search query results

4.4 Discussion

In this section, the research queries defined before in section 4.1 will be answered, taking into consideration the studies found in the Data Collection phase.

4.4.1 RQ1: How does the choice between relational and non-relational databases impact Java-based applications' energy consumption and performance?

The study done in [52] analysed the impact of various databases and cloud patterns on energy consumption and performance in cloud-based applications by conducting hypothesis tests, the metrics used were the energy consumption of CPU and Memory and the Response time. It focused on three different databases: MySQL and PostgreSQL (relational), and MongoDB (NoSQL). The results showed that MySQL is the least energy-consuming database, but also the slowest in terms of performance. PostgreSQL, while faster than MySQL, is the most energy-consuming database. MongoDB, the NoSQL database, consumes more energy than MySQL but less than PostgreSQL, and is the fastest among the three.

The study in [50] conducted experiments in which MongoDB outperformed PostgreSQL in handling large, unstructured datasets with high scalability. The experimental results suggest that MongoDB generally offers better performance in terms of scalability, speed, and efficiency compared to PostgreSQL for certain use cases, especially in large-scale applications with flexible schema requirements. The results obtained consisted of the Response time taken from each database.

In [56], CouchDB and MySQL were evaluated for their performance in CRUD (Create, Read, Update, Delete) operations within Java-based applications, measuring the response time for each one. MySQL was the better option for use cases where the data structure is well-defined, complex relational operations are necessary, or transactional integrity is a priority. However, CouchDB proved to be more suitable for applications requiring high scalability, flexibility, and the ability to handle large volumes of unstructured data.

Research concluded that relational databases have different response time results when compared with each other. The study performed in [49] concluded that PostgreSQL stands out for its strong performance in CREATE and UPDATE operations, particularly when dealing with larger datasets. However, it faces significant challenges when it comes to DELETE operations as the number of records increases. On the other hand, MySQL excels in deletion speed, outperforming other relational database management systems like Oracle by a factor of more than 12.5 times when deleting 500,000 entries. MySQL's READ operations also perform well for

smaller datasets, though PostgreSQL outshines it in this regard for datasets up to 50,000 records. Additionally, in [48], it was studied that the framework used can have an impact on Java's application performance. Hibernate offered high configurability but had performance overheads, while JPA and Spring Data JPA provided better performance for simpler use cases.

The study in [47] found that non-relational databases like MongoDB offer better performance for storing and processing large datasets, especially in applications where data is generated rapidly and can vary in structure. In contrast, relational databases such as MySQL may struggle with large volumes of semi-structured or unstructured data, which is common in environments that require high scalability and flexibility. MongoDB, with its horizontal scalability, can manage growing datasets more efficiently, while MySQL is typically optimised for smaller, more structured datasets with rigid schemas. To analyse the performance, it was measured the throughput of each database (messages per ms) and the scalability with different data volumes.

Conducted experiments in [46] revealed that NoSQL databases generally outperform relational databases in completing data operations, especially when dealing with large datasets. This advantage is primarily attributed to the flexible, schema-free structure of NoSQL databases, which allows them to scale more effectively and process data more rapidly. The experiments demonstrated that even when relational databases store their data in memory, they still fall short in performance when compared to their NoSQL counterparts. This outcome highlights that while in-memory storage can improve relational database performance, it does not match the operational speed and efficiency provided by NoSQL systems. Detailing the comparisons between Memcached, H2, Redis, Cassandra and MongoDB databases:

- Memcached offers the best write performance in terms of elapsed time, meaning it is the fastest for writing data.
- Redis provides better read performance than Memcached and MongoDB.
- H2 has the worst read performance, likely due to its relational database architecture.
- Redis excels at deleting key-value pairs, offering the best performance, while MongoDB has the worst performance for this operation.
- Redis and Cassandra also consume memory more effectively than others for delete operations.
- MongoDB performs best for fetching the entire dataset, while Cassandra is the slowest.
- Redis and MongoDB serve data from memory, using disk storage for persistence, whereas H2 stores data in volatile memory, making it less reliable for large datasets.
- H2 uses more memory than others, especially as the data size grows.

A study [53] was done about the performance between Redis, a key-value database, and MariaDB, a relational database, on big data applications, measuring the response time of each one. They experienced that Redis offers superior runtime performance for insert, delete, and update operations, particularly when updates involve specific conditions, and excels in single-result queries and handling complex queries, due to Redis's flexibility, schema-less structure, and user-friendly design. On the other hand, MariaDB's more complex data model can hinder query performance. However, Redis does have limitations when it comes to routine querying and updates on large datasets.

The performance of PostgreSQL, MongoDB, and Apache Cassandra was analysed in [6] measuring the average execution time in ms. Writing operations performed better in relational databases than in non-relational ones. However, in the context of reading, the non-relational ones outperformed the relational ones, especially MongoDB.

4.4.2 RQ2: How does query optimisation for relational databases affect energy consumption and performance compared to non-relational databases?

The literature review in [54] highlights the role of query optimisation in enhancing the performance and scalability of relational databases like Oracle and MySQL. They both have high-level optimisation capabilities such as indexing, partitioning, and in-memory processing to optimise the performance. They can process complicated queries with minimal response time using these capabilities, and thus, they are viable for high-performance environments. Oracle is especially renowned for its superior management of complicated transactions and scaling, which it does by utilising its high-level optimisation capabilities. MySQL is especially renowned for its simplicity and performance in processing medium workloads. While it may require additional effort to optimise large-scale distributed environments, MySQL remains a cost-effective choice for applications with less demanding query requirements.

The study in [55] conducted experimental comparisons of the performance of CRUD operations on PostgreSQL and MongoDB with and without indexing optimisations. Figure 6, Figure 7, Figure 8 and Figure 9 show the results obtained from them.

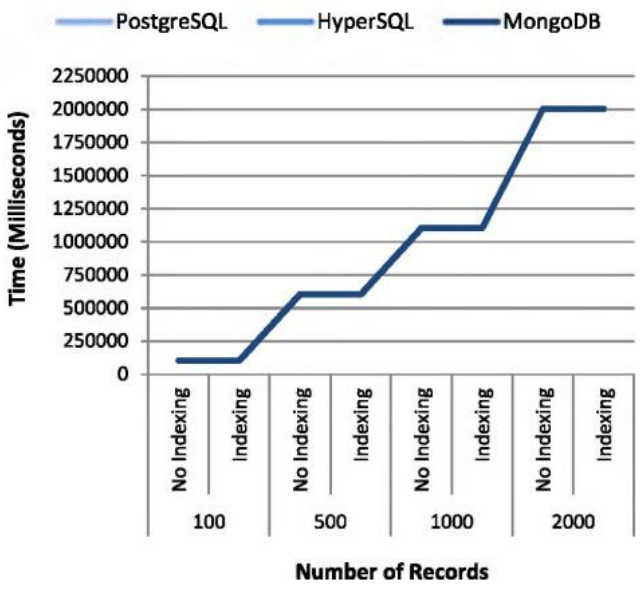


Figure 6 - Insert Operation
Source: [55]

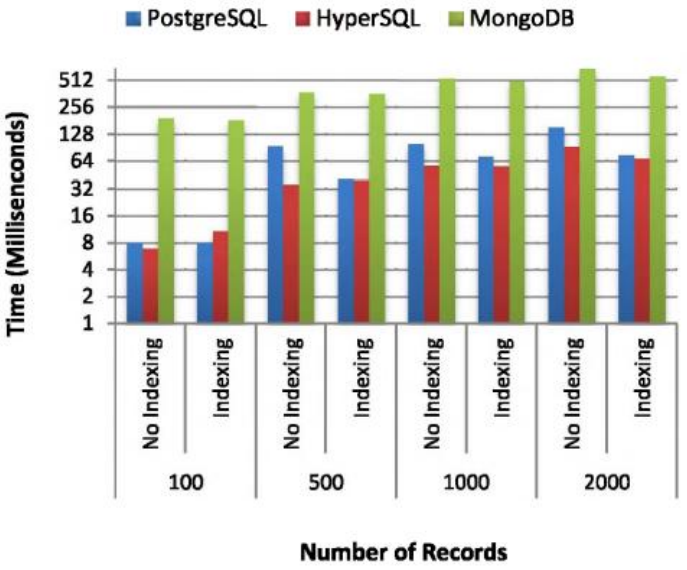


Figure 7 - Select Operation

Source: [55]

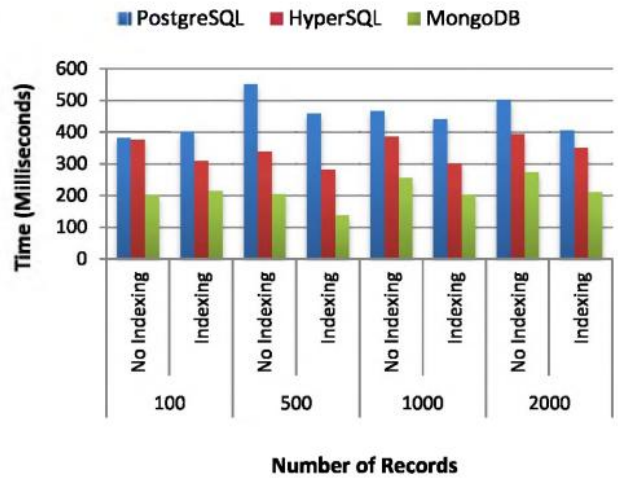


Figure 8 - Update Operation

Source: [55]

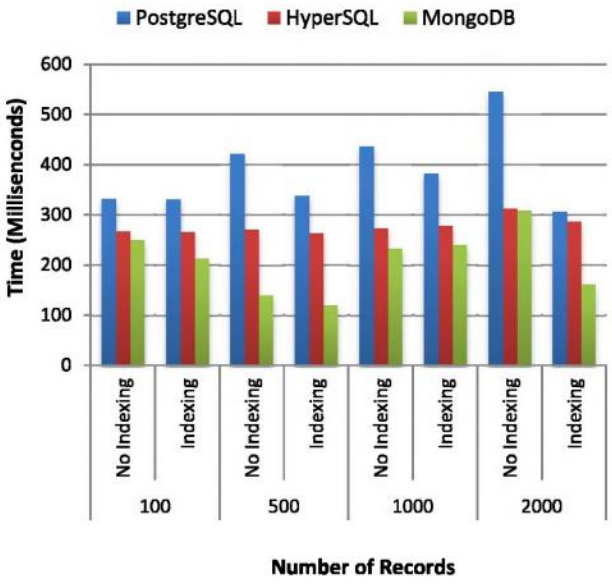


Figure 9 - Delete Operation

Source: [55]

Summarising the information:

- **Insert Operations**
    1. **Without Indexing:**
        1. MongoDB consistently required less time compared to PostgreSQL across all record sizes (100, 500, 1000, 2000 records).
        2. Both databases showed an increase in execution time as the number of records increased, but MongoDB maintained a lower execution time overall.
    2. **With Indexing:**
        1. MongoDB retained its performance advantage, but the time difference between it and PostgreSQL narrowed slightly.
        2. Both databases benefited from indexing, reducing the overall execution time compared to operations without indexing.
- **Select Operations**
    1. **Without Indexing:**
        1. PostgreSQL outperformed MongoDB significantly. The performance gap widened as the number of records increased.
        2. MongoDB's execution times were much higher, highlighting its comparative inefficiency for unindexed SELECT operations.
    2. **With Indexing:**
        1. Indexing improved performance for both databases, with PostgreSQL continuing to show better results. MongoDB, while improved, still showed worse results than PostgreSQL.
- **Update Operations**

1. **Without Indexing:**
    1. MongoDB required less time for updates compared to PostgreSQL across all record sizes.
    2. PostgreSQL showed higher execution times.
2. **With Indexing:**
    1. MongoDB's advantage increased, with significant reductions in execution time. PostgreSQL improved as well, but not enough to surpass MongoDB.

- **Delete Operations**
    1. **Without Indexing:**
        1. MongoDB had better results than PostgreSQL.
    2. **With Indexing:**
        1. PostgreSQL saw improvements with indexing, but MongoDB maintained its superior performance.

To conclude, indexing enhances the performance of both PostgreSQL and MongoDB across all CRUD operations. PostgreSQL demonstrates a marked advantage in SELECT operations with indexing, reflecting its design for structured and optimised queries. In contrast, MongoDB's schema-less architecture enables it to excel consistently in INSERT, UPDATE, and DELETE operations, even when indexing is not applied.

4.4.3 RQ3: How does data volume influence query performance and energy consumption in relational and non-relational databases?

In [58], a study with 3 different datasets was realised:

- D1: Small dataset (estimated 2500 records).
- D2: Medium dataset (estimated 542,000 records).
- D3: Large dataset (estimated 1,189,000 records).

The results gathered included the run-time in seconds and the energy consumption in Joules of CPU and RAM usage. With these results, it was possible to conclude that relational databases generally showed better energy efficiency for smaller datasets and operations like INSERT and JOIN. MySQL, for instance, demonstrated 60.42% greater energy efficiency than Couchbase for UPDATE queries in the smallest dataset (D1). However, as dataset sizes increased (D2 and D3), non-relational databases like MongoDB and Couchbase outperformed SQL systems in energy efficiency. This advantage is attributed to NoSQL systems' advanced caching and indexing mechanisms, which reduce disk I/O operations. Regarding runtime, NoSQL databases were faster for most operations, especially with larger datasets. However, JOIN queries were consistently more runtime-efficient in SQL systems, with PostgreSQL often leading due to its advanced query optimisation techniques. The experiments revealed no single database system consistently excelled in both energy consumption and runtime efficiency across all scenarios. SQL systems were preferable for smaller datasets and operations requiring complex relationships like JOIN. In contrast, NoSQL systems excelled with larger datasets, leveraging scalability and lightweight architectures.

The study in [56] evaluated the impacts of data volume on query performance in MySQL and CouchDB databases, by measuring the response time of each operation with different data sets. For INSERT operations, CouchDB and relational MySQL exhibited similar performance with small datasets (1,000 to 10,000 records). However, as the dataset size increased to 100,000 and 1,000,000 records, CouchDB demonstrated superior performance due to its schema-less design and reduced processing requirements. This advantage became more evident as data volume grew. In UPDATE operations, CouchDB's performance depended on its structure. While its first structure was slower for single and multiple updates, the second structure outperformed relational MySQL as data size increased, thanks to its simplified data handling. For select operations, relational MySQL performed adequately for small datasets but struggled with large ones, especially in complex queries involving multiple joins. CouchDB's second structure scaled better, offering faster response times as dataset sizes grew. In DELETE operations, CouchDB's second structure showed significant advantages for larger datasets, completing deletions more efficiently than relational MySQL, which became slower due to schema and indexing overhead. For smaller datasets, relational MySQL performed slightly better.

In [57], a performance comparison based on response times was done between MySQL, a relational database, and Elasticsearch, a non-relational database, focusing on storing and retrieving key-value pairs, a common use case in e-applications. The results demonstrated that Elasticsearch significantly outperformed MySQL in query execution speed. Specifically, when using Elasticsearch with its Native Java Client, it was approximately five times faster than MySQL operating without an index. Even when MySQL utilised an index on the queried column, Elasticsearch still maintained a clear advantage, being about three times faster. These findings highlight that as data volume grows, non-relational databases like Elasticsearch tend to sustain better query performance compared to relational databases like MySQL. This underscores the efficiency and scalability of Elasticsearch in handling large-scale data operations.

4.5 Conclusion

Relational databases are used when data structures are very well-defined, where complex queries with several joins must be performed, or transactional consistency is very important. With small data sizes, relational databases usually draw more energy efficiency and perform better in INSERT and JOIN operations. However, relational databases are not good at scalability as data size increases, becoming slow and demanding more energy with the rise of large or unstructured data.

On the other hand, non-relational databases show significant capability in managing large data sets with dynamic or evolving schemas. Such databases are likely to have better results than relational databases, both in terms of performance and energy consumption as data increases. The architecture of non-relational databases supports horizontal scaling and enhanced caching effectiveness, reducing disk I/O operations and enhancing runtime performance. For INSERT, UPDATE, and DELETE operations, non-relational databases generally maintain better performance even in the absence of the necessity for thorough query optimisation or indexing.

Further, it can be concluded that relational database query optimisation is the most critical thing, especially in the case of complex queries. Indexing is a method that improves the performance of relational systems to help them handle larger and more complex datasets easily. However, the non-relational databases are still superior in case of humongous amounts of unstructured data, especially for high-throughput and scalable environments.

In summary, the research shows relational databases remain the best for smaller data set applications and complex relational specifications. However, for larger data sets or applications that require flexibility and scalability, non-relational databases scale better and are more efficient, providing better overall performance and energy consumption in these situations. Choosing relational or non-relational databases purely depends on the requirements of the application, precisely in data structure, query complexity, and system scalability.

5 Experimental Design

This chapter enhances the tools and frameworks selected for the experiments in the section 5.1, explaining the reasons for their selection. Also, it references the project chosen to perform these experiments, along with its architectural design, in the section 5.2. Concluding, in the section 5.3, there are five scenarios mentioned that will be used to perform the experiments.

5.1 Selected tools and frameworks

A set of proven tools and frameworks was selected to accomplish the performance, energy consumption, and behaviour experiments described in this study. This set is composed of only three databases due to the limited time of the author. However, the introduction of a key-value store database would be a good addition to the experiment.

Therefore, the three database management systems to compare are MySQL, MongoDB, and Apache Cassandra. They are different paradigms: relational, document-based, and column-oriented, respectively. MySQL and MongoDB are at the top of the most popular databases [59], which is a good reason to study their differences. On the other hand, Apache Cassandra, using a schema structure, can be a good example to get comparison results against MySQL. Also, these databases are open source, and the author has knowledge of them, especially MySQL, which are two major reasons for their selection.

One of the critical demands of the experiments is the insertion of massive quantities of data efficiently. All the selected databases support bulk data insertion. MySQL and Apache Cassandra have imported structured data via CSV files [60], [61]. MongoDB supports bulk insertion via JSON files [62]. Based on the author's experience, these files can be generated using custom scripts developed in Python.

The tools selected from the sections 3.4.2 and 3.5.2 to measure the performance and energy consumption were Grafana k6 and Kepler. This decision was based on Star's qualification and the update rates of JMeter's GitHub [63], Grafana k6's GitHub [64], Kepler's GitHub [65], and JoularJx's GitHub [66]. From these criteria, Grafana k6 and Kepler distinguish themselves from the others.

5.2 Project Selection

To answer and investigate the Research Questions mentioned in the section 4.4, a Java project is required. To mitigate the extra necessary time, one project will be chosen. When selecting a project, some attributes and criteria were considered:

- It must be open-source or have all the code available.

- It should be very well documented.
- The project must be recent, less than 5 years old.
- Have JPA or JDBC connections to a database, preferably JDBC.
- It must be prepared to have a relational and non-relational database.

The author had experienced project development during his Master's Degree study in Software Engineering, which followed some characteristics that match the mentioned criteria:

- Since the author is one of the owners, he has full access to the project's code.
- There is a lot of documentation, mentioning its architecture and database model.
- The project was developed in the same curricular year as this dissertation investigation.
- JPA communicates with the database. It is not JDBC, but if required, the adaptation can be easily done.
- It has an easy-to-understand database model, and the join among tables can be complex enough to compare against non-relational databases.
- The database used was MySQL, and since it is very well structured, the adaptation to other databases is not complex.

The project was also developed with Oracle OpenJDK 23, which is one of the latest versions of Java, on October 15, 2024 [67].

5.2.1 Business Context

During the Master's Degree study in Software Engineering, the author and four colleagues were asked to develop a project to design, build, and present a citizen-focused mobile application that improves interaction with city services through innovation, engagement, and digital empowerment. The application is intended to serve as a platform for urban engagement, allowing citizens to:

- Report urban issues.
- Praise city workers or initiatives, reinforcing positive civic behaviours.
- Participate in community-driven events, fostering a sense of local involvement.

The team chose to create an application that encourages group registration and evaluation of open Wi-Fi spots in urban settings. There are three main functionalities in the application:

- Users register new Wi-Fi locations.
- Users post reviews based on their experiences.
- Users mark visits to existing spots.

The application will include a gamification system where users earn points for these activities to increase engagement and encourage ongoing interaction. Users and nearby businesses can benefit from a mutually beneficial ecosystem as these points can be exchanged for rewards at participating Wi-Fi locations. Furthermore, the platform is intended to facilitate a premium model wherein entrepreneurs function as premium. This gamification system was not fully implemented, although the essential functionalities mentioned were.

5.2.2 Architecture

The project owners documented the project's architecture, although the author altered it to include the non-relational database components. So, the deployment diagram developed is available on Figure 10.

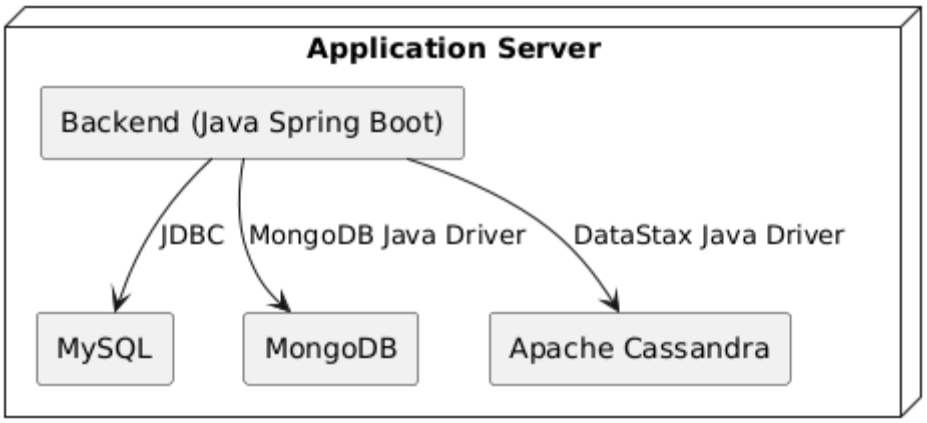


Figure 10 - Deployment Diagram

The application backend is monolithic, hosted on a single server alongside the three databases. These databases are executed within Docker containers on the local machine. This was done after learning from prior use of servers from Departamento de Engenharia Informática (DEI) of Instituto Superior de Engenharia do Porto (ISEP), in which loading millions of records into the database had resulted in severe performance slowdowns and service unreliability.

Some elements were removed from the original deployment diagram:

- The frontend was removed, considering it won't be used for this investigation.
- The AI system was also removed.

An existing component diagram was also altered following the updates mentioned above. This diagram, available in Figure 11, displays the communication between the backend and databases, MySQL, MongoDB and Apache Cassandra. The MySQL connection is handled via JDBC, as outlined in the section 3.3.1. For MongoDB, the system uses the MongoDB Java Driver. Finally, the Apache Cassandra integration employs the DataStax Java Driver.

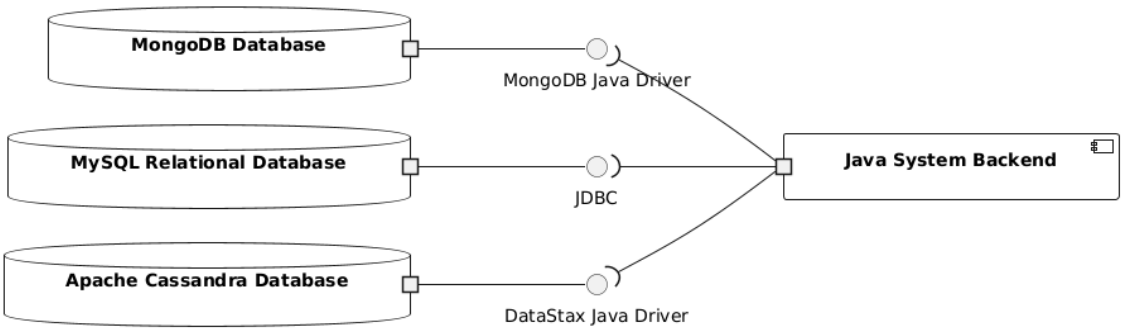


Figure 11 - Component Diagram

5.2.3 Data Model

The domain model is presented in Appendix C, using class diagrams for each aggregate, as it is too long to be included in the document.

However, the data model was adapted for each database, and this section documents the respective data models.

5.2.3.1 Data Model on MySQL

The project's data model of MySQL contains five aggregates:

- **Review** – This aggregate manages the reviews of Wi-Fi spots. It includes an overall rating, a comment, and additional attribute ratings. Reviews are associated with the user who added them and their Wi-Fi spot.
- **Wifi Spot** – This aggregate represents the free Wi-Fi points created by the users. Information about each Wi-Fi spot includes its name, description, coordinates, environmental conditions, technical quality indicators, and available facilities. Wi-Fi hotspots are classified by type (public, café, library, etc.) and management status (verified, unverified, sponsored, etc.). If a Wi-Fi spot is part of a business that offers rewards, it may be associated with a premium user who oversees it and the user who submitted it.
- **Wifi Spot Visit** – This collection is responsible for handling the user visits at Wi-Fi places. A visit includes timestamps for its start and end and is linked to both the user and the Wi-Fi site visited. This data is used in gamification by rewarding users with points and monitoring levels of activity.
- **User** – It contains the data for the users, i.e., the regular and premium users. Roles, depending on which access is given to the features, like providing offers for the premium users, are responsible for this summary as well.
- **Points Earned Transaction** – It handles the points the user will earn from running the application, and they are earned by the user in terms of an initial setup for a Wi-Fi hotspot, in writing a review, or while checking out a location. Points are subsequently

able to be converted into rewards, especially at partner locations where the loyalty scheme of the app offers benefits for access.

5.2.3.2 Data Model on MongoDB

MongoDB is a schema-less, document database. It does not have a strict data model. However, for this project and to enable ease of data model structure in the implementation and interpretation of subsequent performance experiments, a structured data model was intentionally adopted.

Like the data model for MySQL:

- The **User** set contains important user data, including profile data, roles, and addresses.
- The **Wifi Spot** collection holds data about the registered Wi-Fi spots, their geographic coordinates, attributes, and embedded addresses.
- The **Review** collection allows users to comment and rate their experience at a Wi-Fi spot, with the option of an overall rating to user-specified attribute-value pairs. One essential difference against MySQL is that the attributes can be stored with the document, not needing JOINS to retrieve their information.
- The **Points Earned Transaction** collection holds all instances where users gain points by posting reviews, registering spots, or visiting them, with references to the entities involved.

Regarding to **Wifi Spot Visit** collection, two versions were employed during design. The first version consisted of a normalised structure with foreign key references to the user and Wi-Fi spot, and stored only the start and end timestamps of the visit. To improve the utilisation of the dataset and leverage MongoDB's capability to handle denormalised data well, a second version of the collection was later added. Its new version includes embedded data about the used Wi-Fi location, such as its location and name. This denormalised form makes more self-sufficient documents available for faster reads and simpler querying for data analysis, allowing comparative experimentation between normalised and denormalised data access patterns.

5.2.3.3 Data Model on Apache Cassandra

Despite Apache Cassandra not strictly defining its schema as MySQL, it is not schema-less. Three data model possibilities were considered for the Wi-Fi spot visit entity visible in Table 6.

Possibility	Description	Advantages	Disadvantages
Embedded data	Embed wifi_spot_name in the wifi_spot_visit table.	Fast read performance for visit feeds; no joins required.	Redundant data leads to slow updates and risks.
Normalised schema	Store only the wifi_spot_id in wifi_spot_visit; retrieve the name from wifi_spot.	Faster updates, no duplication of spot data, and consistent data integrity.	Requires application-level joins, which increases read latency and implementation complexity.
Partitioned + clustering strategy	Create a visit table partitioned by	Supports efficient ordering by time,	Still suffers from heavy update

	wifi_spot_id, with clustering by visit_start or time.	potentially faster updates, and a query-friendly layout.	volumes and requires careful tuning of clustering and TTLs.

Table 6 - Different data model possibilities for Apache Cassandra

So, the data model chosen was the first possibility that combines aspects from MySQL and MongoDB:

- The **Wifi Spots** address was embedded in the same entity.
- The **Wifi Spot Visit** followed MongoDB's alterations; the Wi-Fi spot information, name and location were embedded in the same entity.
- The **Review** kept the MySQL version by storing its attributes in a separate table.

This decision was primarily driven by the desire to maintain structural similarity with the MySQL schema, thereby reducing the number of changes required and ensuring a fairer comparison across database systems. The aim was to conduct experiments using data models that were as equivalent as possible to isolate performance differences caused by the database engines rather than the schema design.

Additionally, this approach prioritises read-heavy scenarios, particularly complex SELECT queries, which align with the indexing and optimisation strategies defined for the experiments. While this model does lead to performance degradation during UPDATE operations, due to data duplication and denormalisation, this trade-off was considered acceptable.

5.3 Selected scenarios and experiments

During the experiments, five scenarios were selected to study the tradeoffs between performance and energy consumption:

- One insert scenario.
- One update scenario.
- One delete scenario.
- Two select scenarios: simple and complex JOIN queries.

All these scenarios will be tested on three different data sizes, 500, 5000 and 50000:

- **Insert Scenario: Creating a New Review with Associated Attributes** – This is a case of creating a new review for a Wi-Fi spot, with a set of attribute-value pairs describing the experience. In MySQL and Apache Cassandra, the review and its attributes are stored in separate tables, requiring multiple writes and referential integrity. On the other hand, MongoDB puts the review and its fields in a single embedded document and can insert it in a single operation. This will test the performance and energy consumption between embedded documents and normalised models on insert operations.
- **Update Scenario: Updating Wi-Fi Spot Name** – In this scenario, the Wi-Fi Spot name is updated. In MySQL, only the appropriate record in the Wi-Fi Spot table would be

modified. In MongoDB and Cassandra, Wi-Fi spot information is stored within associated documents, so multiple documents need to be updated to maintain the consistency of the data. This test will measure the impact of denormalisation on update performance and energy consumption.

- **Delete Scenario: Deleting a Review and all the attributes associated** – This experiment tests the deletion of a review and all its associated attributes. In MySQL and Cassandra, where attributes are modelled in a separate structure, this operation involves cascading deletes or multiple deletion commands. MongoDB simplifies this process by storing attributes within the review document, allowing a single deletion operation to remove all related data. The experiment will highlight the tradeoffs in deletion performance and energy consumption across the three systems.
- **Simple Select Scenario: Retrieve all reviews for a Wi-Fi spot, ordered from oldest to newest** – This scenario represents a simple SELECT query because it retrieves all the reviews associated with a Wi-Fi spot, requiring only one JOIN operation or any [68] on Apache Cassandra due to the denormalised structure. It allows for testing ordered query execution across the three systems with simple connections between tables.
- **Complex Select Scenario: Build a Review Feed with User and Wi-Fi Spot Info that was visited at least once** – The final scenario simulates the generation of a feed that displays user reviews along with associated user and Wi-Fi spot details that were visited at least once. MySQL requires multiple joins across normalised tables (review, user, wi-fi spots, review attributes, and visits), which makes it a complex query [68] and can be performance-intensive. However, there are no functionalities such as JOIN in Cassandra and MongoDB. This will allow us to study the impact of complex JOIN operations on relational and non-relational databases.

6 Experimental Implementation

This chapter is responsible for explaining the databases implementation, how the bulk data was performed, and the development of the application to be able to analyse the mentioned scenarios, in the sections 6.1, 6.2 and 6.3, respectively. Also, in the section 6.4, it contextualises the implementation of performance and energy consumption measurement tools.

6.1 Databases Configuration

The databases were installed in Docker using three Docker-compose files, referencing MySQL version 8.2 [69], MongoDB 4.4 [70], and Cassandra 5.0 [71], the latest versions as of this date. Each Docker-Compose file can be seen in the project's GitHub [72], [73], [74].

6.2 Bulk Data Creation

Since the application uses databases implemented from scratch, it is necessary to generate big data volumes to test how the data volume affects the results. Therefore, Python scripts were implemented to generate CSV files for MySQL and Apache Cassandra and JSON files for MongoDB.

During the Master's Degree, the author needed to generate this script to create CSV files and do Bulk Inserts in MySQL. So, some work was reused, but now, it was adapted to this application's schema and for JSON files too. The Python package Faker was used to generate fake data for the files [75].

Three scripts were developed to create data files for the three databases [76], [77], [78]. To simulate a real scenario, the wifi spot visits created for each wifi spot were randomly generated from 0 to 1/1000 of the total data size, and the review attributes were also randomly generated from 2 to 5 for each review.

6.3 Application

The application has five endpoints that correspond to the five scenarios mentioned in the section 5.3:

- Reviews creation – This endpoint reads the data from the file and creates objects of Review DTOS and calls its service, which calls its repository to make the reviews.
- Update Wifi spot name – It reads all the data from the script file and updates the Wi-Fi spots with the name on them.

- Delete reviews – Deletes all the reviews in the file.
- Query all reviews from one wifi spot ordered by descending creation date.
- Query all reviews in the system that were visited at least once.

These five endpoints were multiplied by three to distinguish the connections for each database, MySQL, MongoDB and Apache Cassandra.

6.3.1 MySQL

The data access layer for MySQL was built in JDBC, using libraries from the *java.sql* package. The connection to the database is performed with the *DriverManager.getConnection*. The SQL statements were performed using the *PreparedStatement* class, and their parameters were set using methods such as *setString()*, *setInt()*, among others. At the end, the statements are executed using the *executeUpdate()* method or *executeQuery()* for the SELECT methods. On the Code Snippet 8 there is an example of a SQL statement created.

```
public int updateWifiSpotName(UUID wifiSpotId, String newName) {
    String sql = "UPDATE wifi_spot SET wifi_spot_name = ? WHERE
wifi_spot_id = UUID_TO_BIN(?)";

    try (Connection connection = DriverManager.getConnection(url,
username, password);
         PreparedStatement stmt = connection.prepareStatement(sql))
{

        stmt.setString(1, newName);
        stmt.setString(2, wifiSpotId.toString());
        return stmt.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException("Error updating Wi-Fi spot name",
e);
    }
}
```

Code Snippet 8 - Update Wifi Spot Name – MySQL

For the SELECT operations, the same process was used but with JOIN operators to connect with other tables and a *ResultSet* object is used to get the statement's result, like the Code Snippet 9 shows.

```
public List<ReviewDto> getReviewsForWifiSpot(UUID wifiSpotId) {
    String sql = """
    SELECT
        BIN_TO_UUID(r.review_id) AS review_id, r.review_comment,
r.review_create_date_time, r.review_overall_classification,
        BIN_TO_UUID(r.review_wifi_spot_id) AS wifi_spot_id,
BIN_TO_UUID(r.review_user_id) AS user_id, u.user_name
    FROM review r
    JOIN users u ON r.review_user_id = u.user_id
    WHERE r.review_wifi_spot_id = UUID_TO_BIN(?)
    ORDER BY r.review_create_date_time ASC
""";

    try (Connection connection = DriverManager.getConnection(url,
username, password);
         PreparedStatement stmt = connection.prepareStatement(sql))
{

        stmt.setString(1, wifiSpotId.toString());
        ResultSet rs = stmt.executeQuery();
    }
}
```

Code Snippet 9 – Select Reviews of a Wifi Spot - MySQL

6.3.2 MongoDB

The data access layer for MongoDB uses packages from *com.mongodb.client* and *org.bson*. The database connection is established using the *MongoClient* class, which is injected into the repository by a customised class [79]. Since the code is using *UUID*, it was necessary to define the *uuidRepresentation* as STANDARD, otherwise the IDs serialisation would not be done in a binary format, like the other databases do.

With the connection, it is called the method *getCollection* to have a *MongoCollection* ready to execute the operations, referenced in Code Snippet 10.

```
public WifiSpotRepositoryMongoDB(MongoClient mongoClient) {
    MongoDatabase database = mongoClient.getDatabase("netQuest");
    this.collection = database.getCollection("wifi_spot");
    this.wifiSpotVisitCollection =
database.getCollection("wifi_spot_visit");
}
```

Code Snippet 10 - Get Collections of MongoDB

Finally, having the connection ready to perform the operations, it uses the *Filters* and *Updates* classes to create the statement and is executed on the respective collections with the *updateOne* and *updateMany* methods. Code Snippet 11 shows an example of a statement responsible for updating the wifi spot name and its visits. It also uses the methods *createOne* and *deleteOne* when necessary.

```
public void updateWifiSpotName(UUID wifiSpotId, String newName) {

    collection.updateOne(
            Filters.eq("_id", wifiSpotId),
            Updates.set("name", newName)
    );

    wifiSpotVisitCollection.updateMany(
            Filters.eq("wifi_spot._id", wifiSpotId),
            Updates.set("wifi_spot.name", newName)
    );
}
```

Code Snippet 11 - Update Wifi Spot name – MongoDB

For the SELECT operations, it uses the MongoDB aggregation framework that applies filters built with *match*, *lookup*, and *unwind* operators. It is visible in the Code Snippet 12 that a statement was created to query all the reviews of a wifi spot.

```
public List<ReviewDto> getReviewsForWifiSpot(UUID wifiSpotId) {
    List<Bson> pipeline = List.of(
            match(Filters.eq("wifi_spot_id", wifiSpotId)),
            lookup("users", "user_id", "_id", "user_info"),
            unwind("$user_info"),
            sort(Sorts.ascending("create_date_time"))
    );

    List<ReviewDto> reviews = new ArrayList<>();
    for (Document doc : collection.aggregate(pipeline)) {
    }
}
```

Code Snippet 12 - Select Reviews of a Wifi Spot - MongoDB

6.3.3 Apache Cassandra

For Cassandra, the database access layer uses classes from the *com.datastax.oss.driver.api.core* package, for example, *CqlSession*, *PreparedStatement*, *ResultSet*, and *Row*. Database connection is made using a *CqlSession* object, which is injected into the repository. Similarly to MySQL, a *PreparedStatement* binds its parameters and executes them in the session. It can be seen in the Code Snippet 13.

```
public WifiSpotRepositoryCassandra(CqlSession session) {
    this.session = session;
}

public void updateWifiSpotName(UUID wifiSpotId, String newName) {
    String updateSpotQuery = "UPDATE wifi_spot SET wifi_spot_name = ?
WHERE wifi_spot_id = ?";
    session.execute(session.prepare(updateSpotQuery).bind(newName,
wifiSpotId));

    String selectVisits = "SELECT visit_id FROM wifi_spot_visit WHERE
wifi_spot_id = ?";
    PreparedStatement selectStmt = session.prepare(selectVisits);
    ResultSet resultSet =
session.execute(selectStmt.bind(wifiSpotId));

    String updateVisitQuery = "UPDATE wifi_spot_visit SET
wifi_spot_name = ? WHERE visit_id = ?";
    PreparedStatement updateVisitStmt =
session.prepare(updateVisitQuery);

    for (Row row : resultSet) {
        session.execute(updateVisitStmt.bind(
                newName,
                row.getUuid("visit_id")
        ));
    }
}
```

Code Snippet 13 - Update Wifi Spot name - Cassandra

One particularity of Cassandra is that updates with conditions can only be performed using the primary key. So, it was necessary to query all the visits of the wifi spot and then for each visit update the wifi spot name on it. However, to query all the visits from one wifi spot, it was necessary to create an index, with the following command: *CREATE INDEX ON netquest.wifi_spot_visit(wifi_spot_id).*

For the review SELECT queries, it was also necessary to create an index to filter by wifi spot, *CREATE INDEX ON netquest.review(wifi_spot_id).*

6.4 Performance and Energy Consumption

The Grafana k6 and Kepler were installed in the project using command lines, scripts and virtual machine resources. All the steps required for each implementation are detailed in Appendix D and Appendix E, respectively.

7 Analysis of results

This chapter firstly explains the methodology used to gather results about performance and energy consumption through a Goal Question Metric in the section 7.1. Afterwards, in the section 7.2, it documents the analysis of the results obtained from the tools mentioned in 6.4, followed by elaborate hypothesis tests in the section 7.3. Finally, it shows conclusions about the energy and performance tradeoffs in the section 7.4.

7.1 Methodology

To analyse the results and conclude, a Goal Question Metric, GQM, is used. Goal Question Metric is a goal-driven, structured method for defining and selecting metrics appropriate to a specific measurement or prediction scenario [80]. It fosters a top-down strategy in which metrics are directly derived from well-defined goals. Originally developed in the 1980s by Weiss and Basili, and later refined by Rombach, GQM addresses the problem of balancing data collection against organisational goals and interpretation needs [80].

The process begins with the identification of a specific goal, which is subsequently refined through a set of questions designed to make the goal achievable. Suitable metrics are then defined for one or more questions to enable quantification or qualitative measurement [80]:

- Goal – A generic sentence that specifies what to evaluate or enhance, with the aim, object, issue of quality, and viewpoint [81].
- Question – A precise question founded on the objective, used to get certain properties of the object and see whether the goal is being achieved [81].
- Metric – A number or quality measure related to a question, providing the information to answer it and gauge goal fulfilment [81].

Therefore, the defined goal for the analysis of the results of this project investigation is presented in Table 7.

Goal	Define the Tradeoffs between Performance and Energy Consumption for relational and non-relational databases for each operation.

Table 7 - Goal specification – GQM

The questions and metrics defined to analyse the performance results are specified in Table 8.

Question	**Metrics**
Q1: What is the performance of each operation in relational databases?	Average response time (seconds)
Q2: What is the performance of each operation in non-relational databases?	

Table 8 - Performance Questions and Metrics – GQM

The average response time represents the primary performance indicators, giving an overall assessment of the efficiency of its operations. With this metric from Grafana k6, the performance analysis can be performed correctly.

The questions and metrics defined to analyse the energy consumption of the relational and non-relational databases are shown in Table 9.

Question	Metrics
Q3: What is the energy consumption of each operation in relational databases?	Energy Consumption (Joules)
Q4: What is the energy consumption of each operation in non-relational databases?	

Table 9 - Energy Consumption Questions and Metrics – GQM

A measurement scale will not be chosen because the point of the study is to compare the three databases and not decide if it is fast or slow. Also, the environmental resources are not suitable for making those conclusions.

7.2 Analysis

This section is responsible for addressing the performance and energy results obtained from the tools.

7.2.1 Environmental Resource

All experiments and evaluations described in this work were conducted in a personal computing environment with the following specifications:

- **Processor:** 12th Gen Intel(R) Core(TM) i7-1260P @ 2.10 GHz.
- **RAM:** 16.0 GB.
- **System Type:** 64-bit operating system with an x64-based processor.
- **Operating System:** Microsoft Windows.

7.2.2 Performance

This section presents the performance results given by Grafana k6 for the different scenarios and data volumes.

7.2.2.1 Insert Scenario

The following tables, Table 10, Table 11 and Table 12, store the performance results for the Insert scenario, the creation of a review.

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	44.11	43.85	39.69	37.17	36.8	40.324
MongoDB	0.39438	0.18436	0.09494	0.08086	0.07526	0.16596
Apache Cassandra	3.22	5.31	3.99	4.11	8.12	4.95

Table 10 - Performance results for the insert scenario with data volume 500

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	210	186	202	225	225	209.6
MongoDB	4.3	2.58	2.65	1.62	3.13	2.856
Apache Cassandra	25.95	21.64	20.76	25.52	29.47	24.668

Table 11 - Performance results for the insert scenario with data volume 5000

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	1981	1990	2017	1996	2011	1999
MongoDB	30.37	38.86	32.03	37.23	37.24	35.146
Apache Cassandra	181	406	263	274	258	276.4

Table 12 - Performance results for the insert scenario with data volume 50000

An analysis of the INSERT performance across the tested databases highlights considerable differences in how each system handles increasing data volumes. MongoDB demonstrated the best overall performance, maintaining low response times even as the number of records grew significantly. With 500 records, MongoDB completed INSERT operations in an average of approximately 0.17 seconds, while Apache Cassandra averaged just under 5 seconds, and MySQL was notably slower at over 40 seconds, which goes in contrast to what was studied in the section 4.5 that mentioned for INSERT operations and small data sets, MySQL was preferable. This trend remained consistent with a volume of 5,000 records: MongoDB continued to perform efficiently with an average of 2.86 seconds, compared to 24.67 seconds for Cassandra and 209.6 seconds for MySQL.

The disparity widened substantially with the largest dataset of 50,000 records. MongoDB remained performant with an average response time of 35.146 seconds, whereas Apache Cassandra required 276.4 seconds, and MySQL's performance degraded drastically to 1999 seconds. These results suggest that MongoDB is particularly well-suited to high-volume, write-intensive workloads. In contrast, MySQL struggles under heavy insert loads, likely due to its transactional overhead and rigid schema structure. Apache Cassandra offers a more scalable alternative than MySQL, benefiting from its distributed architecture, but it still falls behind MongoDB in raw insert speed. Overall, MongoDB emerges as the most efficient and scalable solution for insert-heavy scenarios across all tested volumes.

Figure 12, Figure 13, and Figure 14 show the disparity between the databases on the different data volumes, and it is possible to understand that its disparity doesn't vary from data volumes, which can indicate that the data volume doesn't influence each database's behaviour.

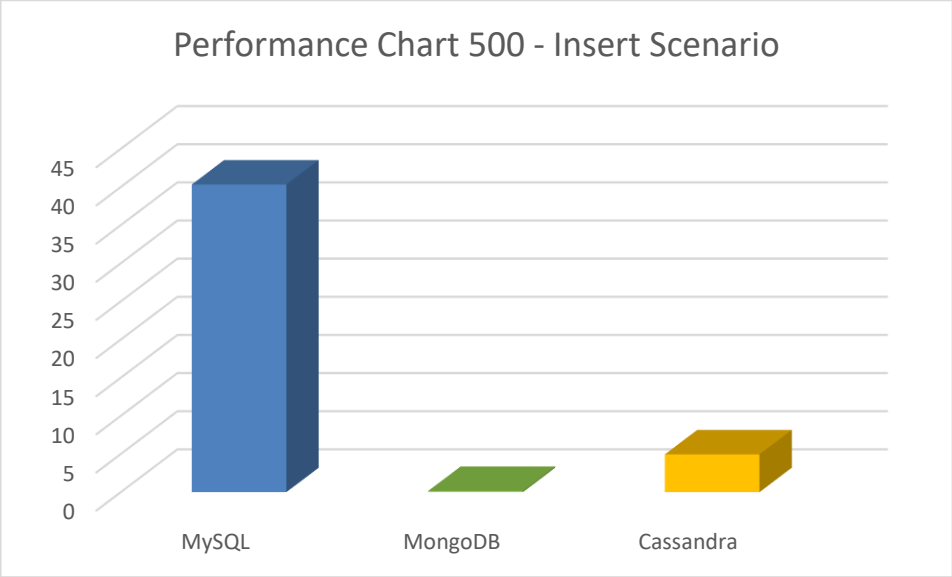


Figure 12 - Performance Chart - Insert Scenario - Volume 500

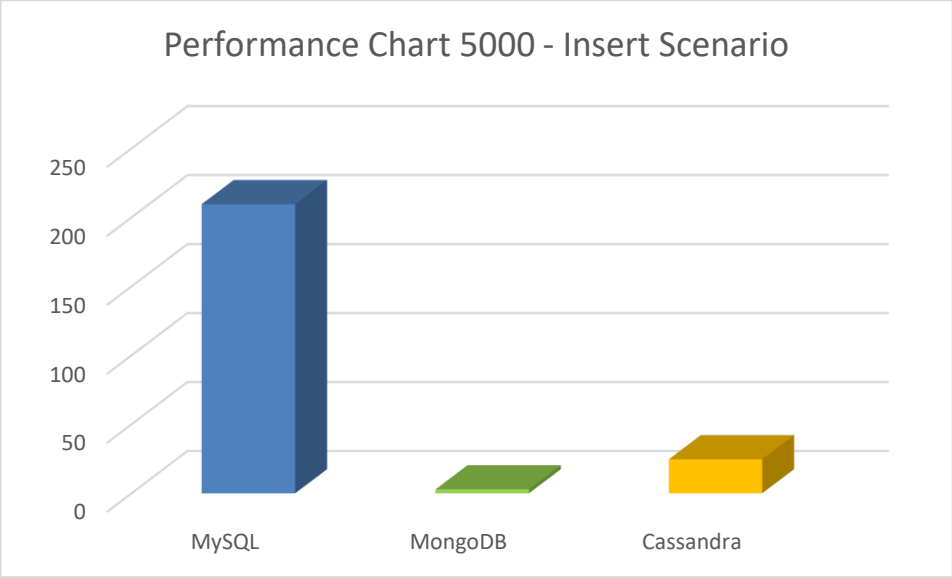


Figure 13 - Performance Chart - Insert Scenario - Volume 5000

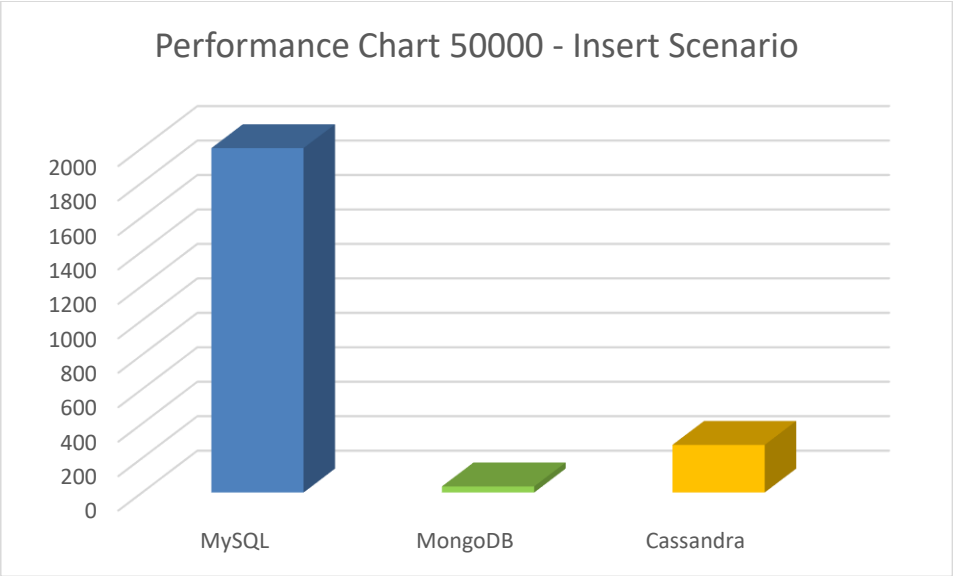


Figure 14 - Performance Chart - Insert Scenario - Volume 50000

7.2.2.2 Update Scenario

The performance results for the Update scenario, updating the Wi-Fi spot name, can be seen in Table 13, Table 14 and Table 15.

Database	**Response Time (seconds)**					
	Executions (1 to 5)					**Average**
MySQL	4.9	4.87	6.83	5.48	6.08	5.632
MongoDB	0.94656	0.64947	0.55132	0.47408	0.53516	0.629318
Apache Cassandra	3.91	4.23	2.36	4.85	3.62	3.794

Table 13 - Performance results for the update scenario with data volume 500

Database	**Response Time (seconds)**					
	Executions (1 to 5)					**Average**
MySQL	107	99	64	62	73	81
MongoDB	64	59.77	59.1	56.33	70	61.84
Apache Cassandra	76	64	62	69	68	67.8

Table 14 - Performance results for the update scenario with data volume 5000

Database	**Response Time (seconds)**					
	Executions (1 to 5)					**Average**
MySQL	545	550	570	602	540	561.4
MongoDB	1654	1703	1651	1655	1661	1664.8
Apache Cassandra	379	462	414	421	385	412.2

Table 15 - Performance results for the update scenario with data volume 50000

The performance evaluation for the update scenario reveals varying behaviours across the three databases as data volume scales. With a small dataset of 500 records, MongoDB was the fastest, averaging just 0.63 seconds, followed by Apache Cassandra at 3.79 seconds, while

MySQL was the slowest at 5.63 seconds. However, this ranking shifted as the dataset grew. At 5000 records, Cassandra overtook MongoDB, achieving an average of 61.84 seconds, ahead of MongoDB's 67.8 seconds and MySQL's 81 seconds. At the highest volume of 50000 records, Cassandra retained its lead with 412.2 seconds, while MySQL came in second at 561.4 seconds, and MongoDB's performance degraded significantly, reaching 1664.8 seconds.

These results show that MongoDB excels at low-volume UPDATE operations but does not scale well for update-heavy workloads. In contrast, Apache Cassandra demonstrates more stable performance growth with increasing volume. MySQL consistently lags at lower volumes but proves more stable than MongoDB when the volume increases, suggesting that its update performance degrades more gradually. These conclusions can be more easily understood from Figure 15, Figure 16, and Figure 17.

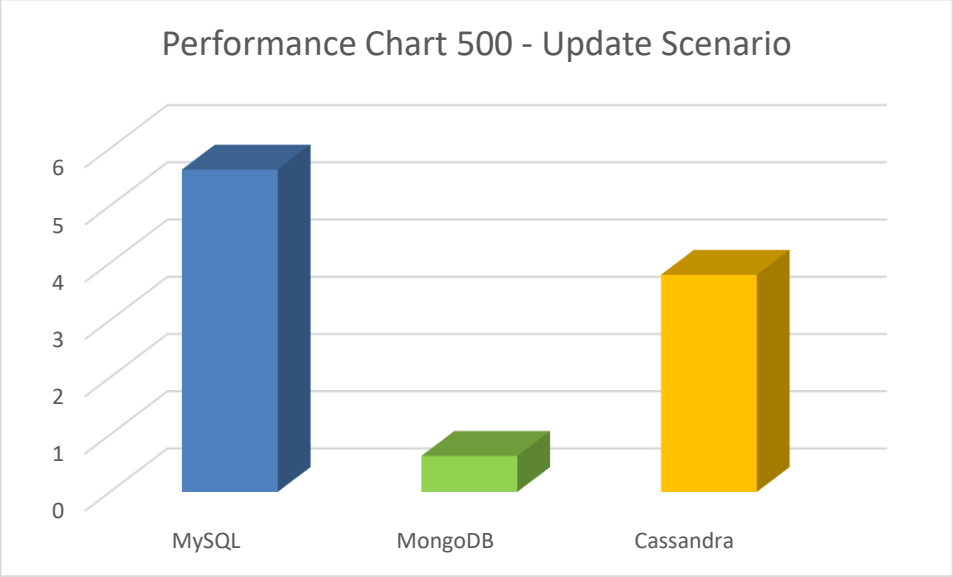


Figure 15 - Performance Chart - Update Scenario - Volume 500

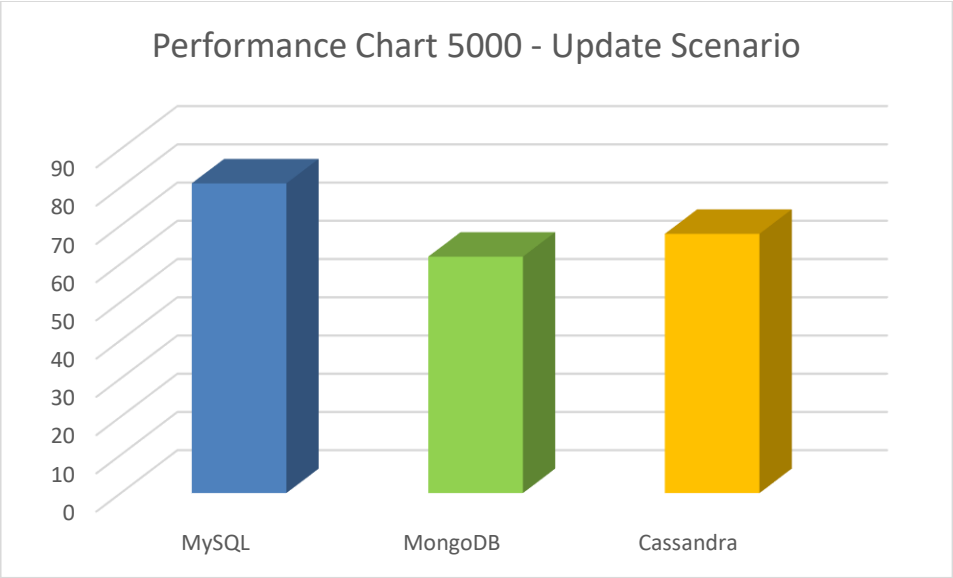


Figure 16 - Performance Chart - Update Scenario - Volume 5000

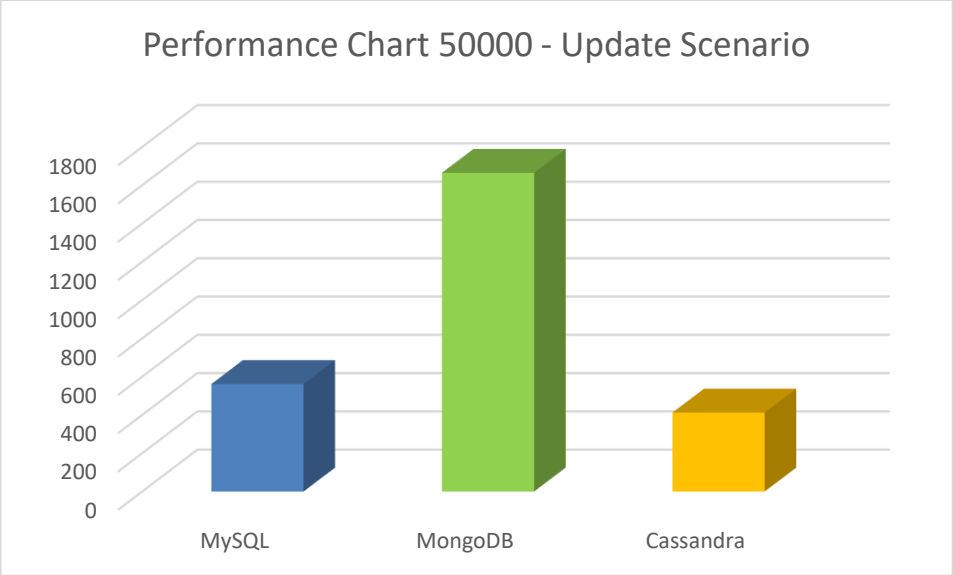


Figure 17 - Performance Chart - Update Scenario - Volume 50000

7.2.2.3 Delete Scenario

The following tables, Table 16, Table 17 and Table 18, show the performance results for the Delete scenario.

Database	**Response Time (seconds)**					
	Executions (1 to 5)					**Average**
MySQL	15.06	18.07	31.66	20.14	26.82	22.35
MongoDB	0.25427	0.23923	0.1162	0.18484	0.19802	0.198512
Apache Cassandra	1.58	1.55	2.02	1.65	1.04	1.568

Table 16 - Performance results for the delete scenario with data volume 500

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	248	280	216	236	246	245.2
MongoDB	4.67	2.76	1.34	4.12	1.79	2.936
Apache Cassandra	22.37	22.1	25.63	21.08	24.72	23.18

Table 17 - Performance results for the delete scenario with data volume 5000

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	2449	2435	2470	2463	2466	2456.6
MongoDB	22.45	31.39	34.31	13.41	23.82	25.076
Apache Cassandra	161	172	190	280	213	203.2

Table 18 - Performance results for the delete scenario with data volume 50000

The delete scenario results demonstrate clear differences in performance among the three database systems, particularly as data volume increases. MongoDB consistently provided the best response times across all scales. For 500 records, it achieved an exceptionally low average response time of 0.198 seconds, compared to 1.57 seconds for Apache Cassandra and 22.35 seconds for MySQL. At 5000 records, MongoDB remained significantly faster, averaging just 2.94 seconds, whereas Cassandra required 23.18 seconds and MySQL lagged with 245.2 seconds. The same pattern held at 50000 records: MongoDB completed the deletions in just 25.076 seconds on average, while Cassandra took 203.2 seconds, and MySQL's performance dropped substantially, with an average of 2456.6 seconds.

These results confirm MongoDB as the most performant database for DELETE operations, especially under increasing data loads. It allows for graceful scaling while maintaining low latency. Cassandra also performs reasonably well, particularly at lower volumes, but its performance deteriorates with larger datasets. MySQL, on the other hand, struggles considerably with DELETE operations as data volume grows. Overall, MongoDB proves to be the most suitable option for applications requiring fast and scalable delete operations. The Figure 18, Figure 19, and Figure 20 provide a graphical view format in which it is possible to obtain these conclusions, similar to the Insert Scenario in the section 7.2.2.1.

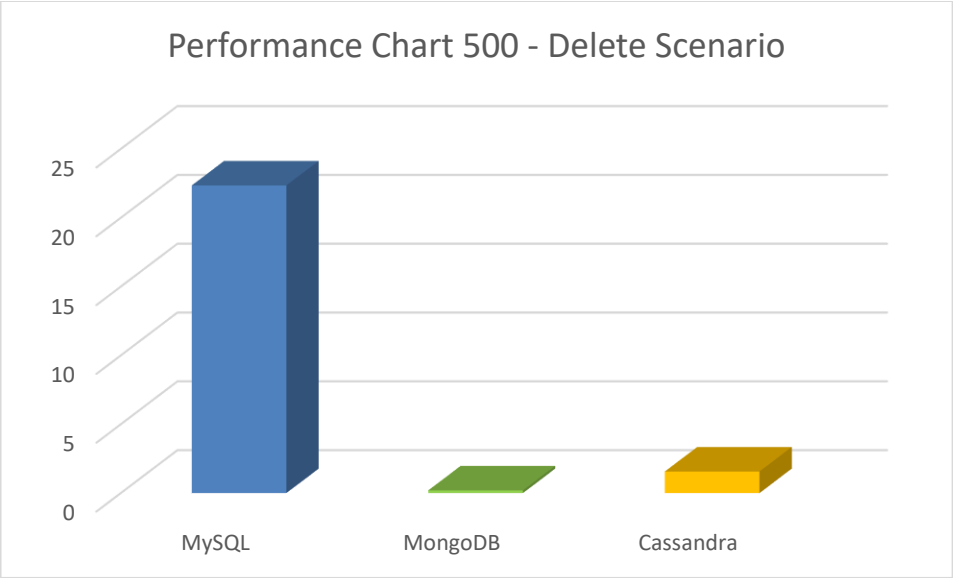


Figure 18 - Performance Chart - Delete Scenario - Volume 500

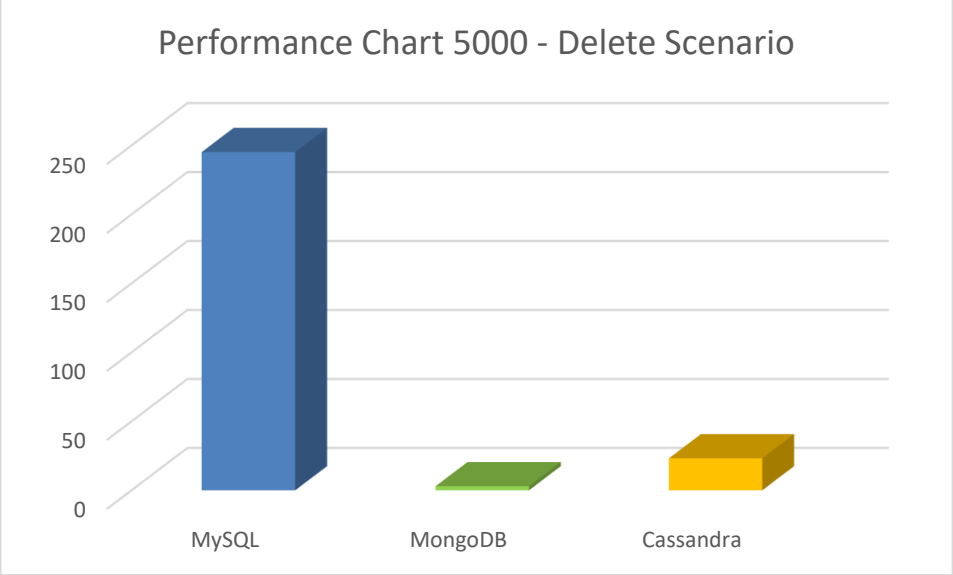


Figure 19 - Performance Chart - Delete Scenario - Volume 5000

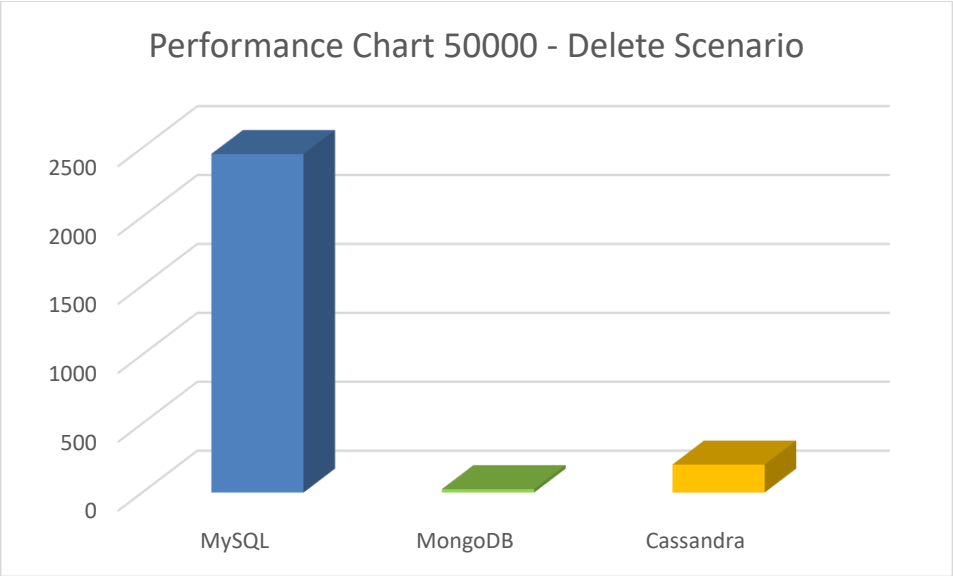


Figure 20 - Performance Chart - Delete Scenario - Volume 50000

7.2.2.4 Simple Select Scenario

Table 19, Table 20 and Table 21 have the performance results of the simple select scenario.

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	0.04523	0.04371	0.03263	0.04596	0.05409	0.044324
MongoDB	0.01537	0.0188	0.0249	0.01356	0.00986	0.016498
Apache Cassandra	0.03154	0.02602	0.02922	0.04344	0.02702	0.031448

Table 19 - Performance results for the simple select scenario with data volume 500

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	0.06909	0.04418	0.04122	0.06758	0.0521	0.054834
MongoDB	0.02289	0.03174	0.01593	0.01818	0.01915	0.021578
Apache Cassandra	0.0286	0.03115	0.00967	0.01485	0.01972	0.020798

Table 20 - Performance results for the simple select scenario with data volume 5000

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	0.03251	0.07706	0.02643	0.02225	0.03657	0.038964
MongoDB	0.03489	0.03404	0.05863	0.04543	0.02736	0.04007
Apache Cassandra	0.11066	0.02888	0.03462	0.02287	0.05122	0,04965

Table 21 - Performance results for the simple select scenario with data volume 50000

The performance evaluation for the simple select scenario reveals consistently low response times across all three database systems, with MongoDB and Apache Cassandra generally outperforming MySQL. At the smallest data volume of 500 records, MongoDB exhibited the best average response time at approximately 0.0165 seconds, followed closely by Cassandra at

0.0314 seconds, while MySQL was slightly slower at 0.0443 seconds, as it is illustrated in Figure 21. With 5000 records, MongoDB remained the most efficient, averaging just over 0.0215 seconds, while Cassandra and MySQL followed with similar averages of approximately 0.0208 and 0.0548 seconds, respectively, as graphically seen in Figure 22. Notably, at the highest data volume of 50000 records, MySQL and MongoDB demonstrated very similar average response times, 0.039 and 0.04 seconds, respectively, observed in Figure 23. While Cassandra's performance degraded somewhat to 0.04905 seconds.

The differences in results are not big enough to conclude that the databases have different behaviours in this scenario; the results are very similar to each other, and the disparity can be due to experimental environment effects.

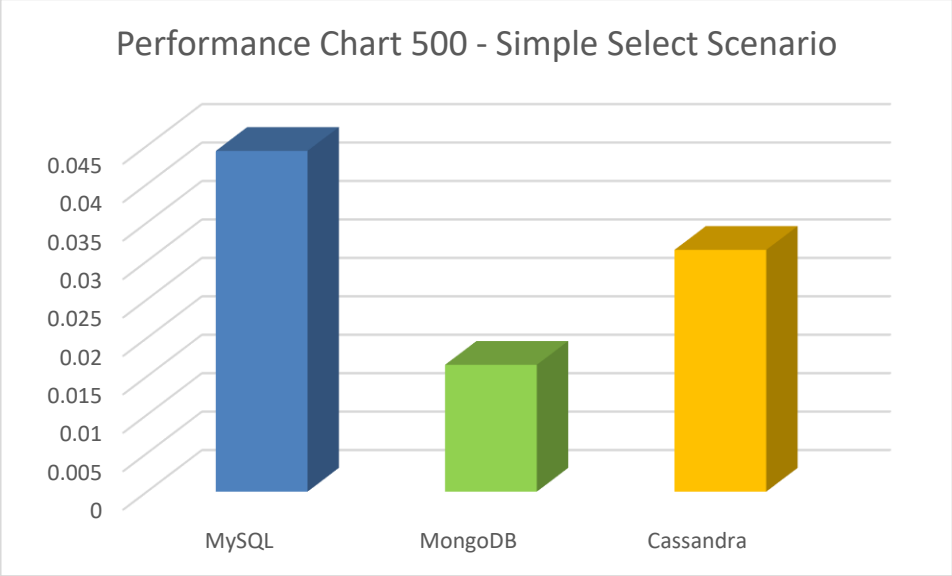


Figure 21 - Performance Chart - Simple Select Scenario - Volume 500

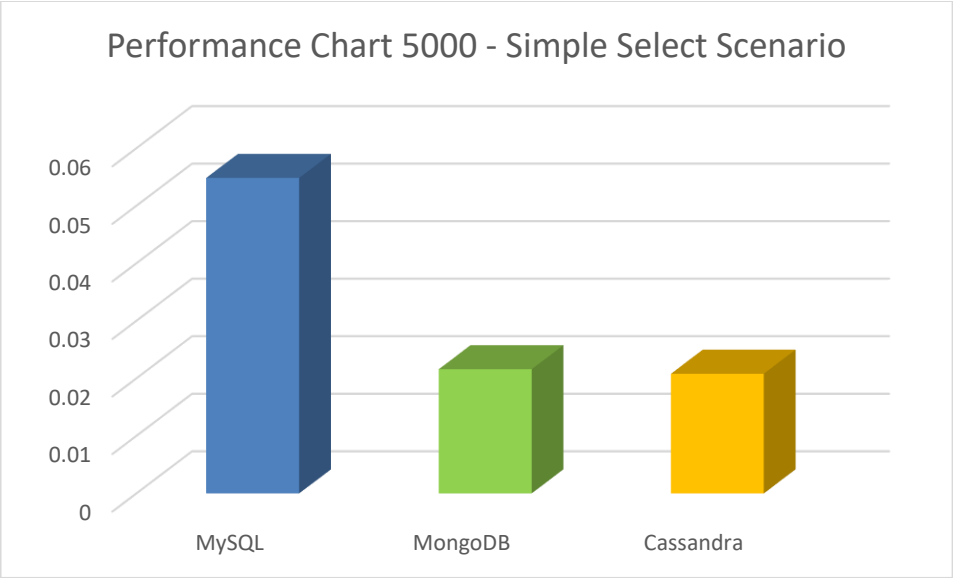


Figure 22 - Performance Chart - Simple Select Scenario - Volume 5000

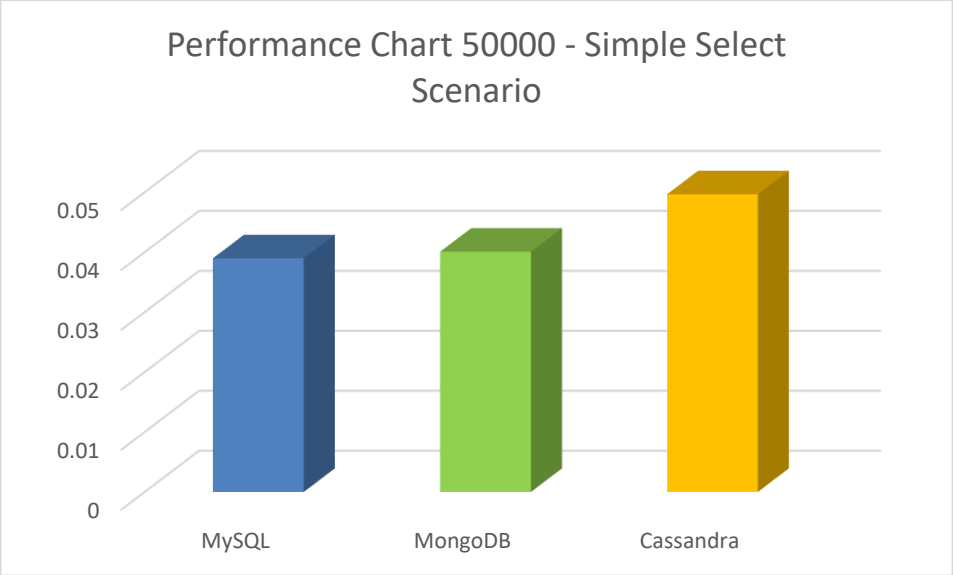


Figure 23 - Performance Chart - Simple Select Scenario - Volume 50000

7.2.2.5 Complex Select Scenario

The following tables, Table 22, Table 23 and Table 24, store the performance results for the complex select scenario.

Database	**Response Time (seconds)**					
	Executions (1 to 5)					**Average**
MySQL	4.11	7.81	4.12	7.27	5.75	5.812
MongoDB	1.98	2.36	1.72	1.37	2.66	2.018
Apache Cassandra	10.24	7.66	6.1	6.59	5.82	7.282

Table 22 - Performance results for the complex select scenario with data volume 500

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	84	65	61	63	66	67.8
MongoDB	66	52.91	64	54.87	61	59.756
Apache Cassandra	76	63	61	62	53.91	63.182

Table 23 - Performance results for the complex select scenario with data volume 5000

Database	Response Time (seconds)					
	Executions (1 to 5)					Average
MySQL	645	681	675	652	673	665.2
MongoDB	1689	1715	1688	1704	1692	1697.6
Apache Cassandra	431	479	482	465	529	477.2

Table 24 - Performance results for the complex select scenario with data volume 50000

The results for the complex select scenario reveal greater variability in performance among the three databases, especially as the data volume increases. At 500 records, MongoDB provided the lowest average response time at 2.018 seconds, followed by MySQL at 5.812 seconds, while Apache Cassandra was significantly slower with an average of 7.282 seconds. With 5000 records, MongoDB continued to lead with an average of 59.756 seconds, although its margin over MySQL (67.8 seconds) and Cassandra (63.182 seconds) was reduced. However, at the highest tested volume of 50000 records, MongoDB's performance degraded sharply to 1697.6 seconds on average, making it the slowest system in this scenario. In contrast, Apache Cassandra handled the large volume more effectively, averaging 477.2 seconds, and MySQL remained slightly behind with 665.2 seconds.

These findings, graphically available in Figure 24, Figure 25, and Figure 26, suggest that while MongoDB excels in processing smaller or moderately sized complex queries, its performance does not scale as efficiently under heavy complex read loads. Apache Cassandra, on the other hand, demonstrates more stable performance as the dataset grows. MySQL, though generally less performant at lower volumes, manages to maintain consistent execution times as the data grows due to its available JOIN operations, remaining competitive with Cassandra at the largest scale. Overall, MongoDB proves optimal for low to moderate volumes in complex queries, whereas Apache Cassandra and MySQL become more suitable choices for handling large-scale complex select operations, which goes against what the literature review concluded, in the section 4.4.3, that the non-relational databases (CouchDB) used performed better as data size increased, which allows us to conclude that even between non-relational databases, there is disparity on their behaviour.

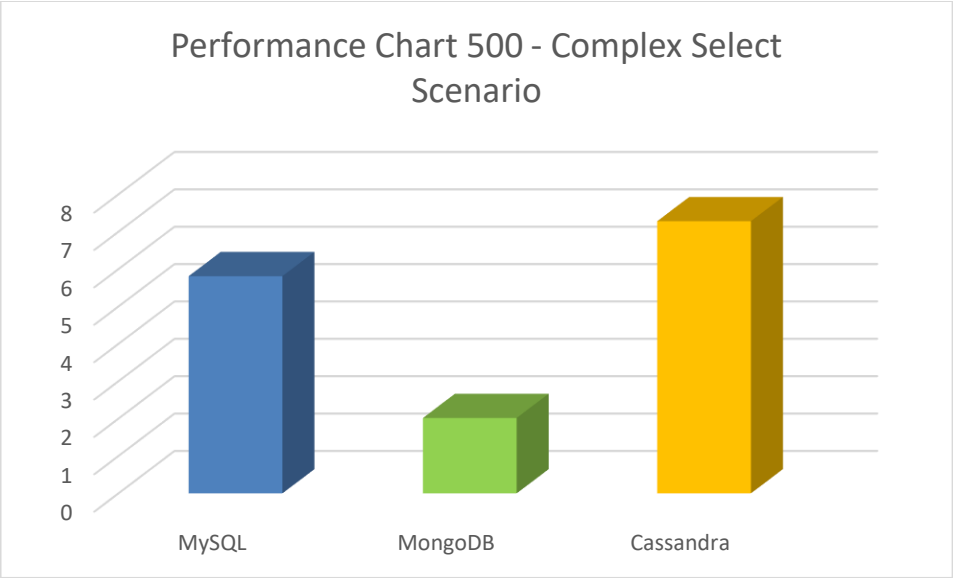


Figure 24 - Performance Chart - Complex Select Scenario - Volume 500

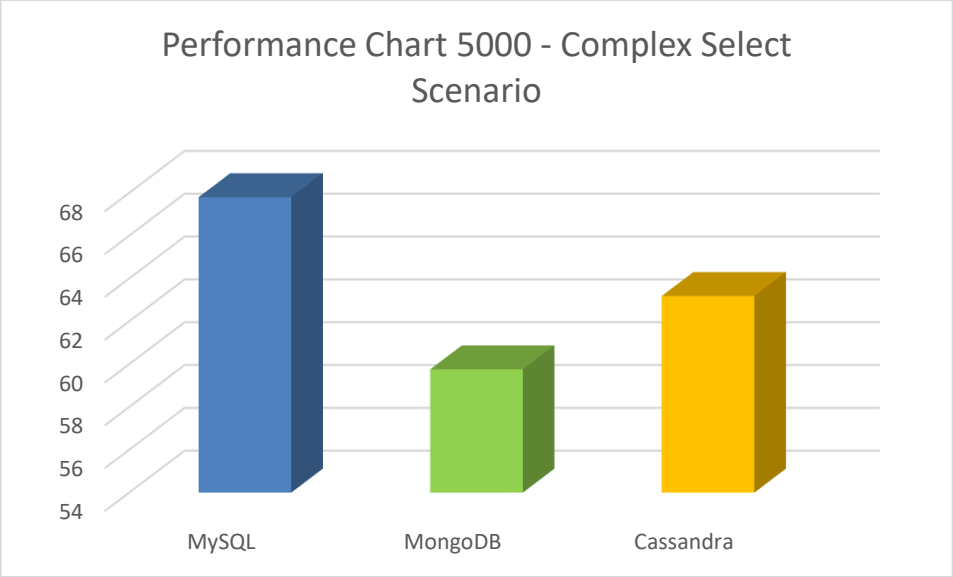


Figure 25 - Performance Chart - Complex Select Scenario - Volume 5000

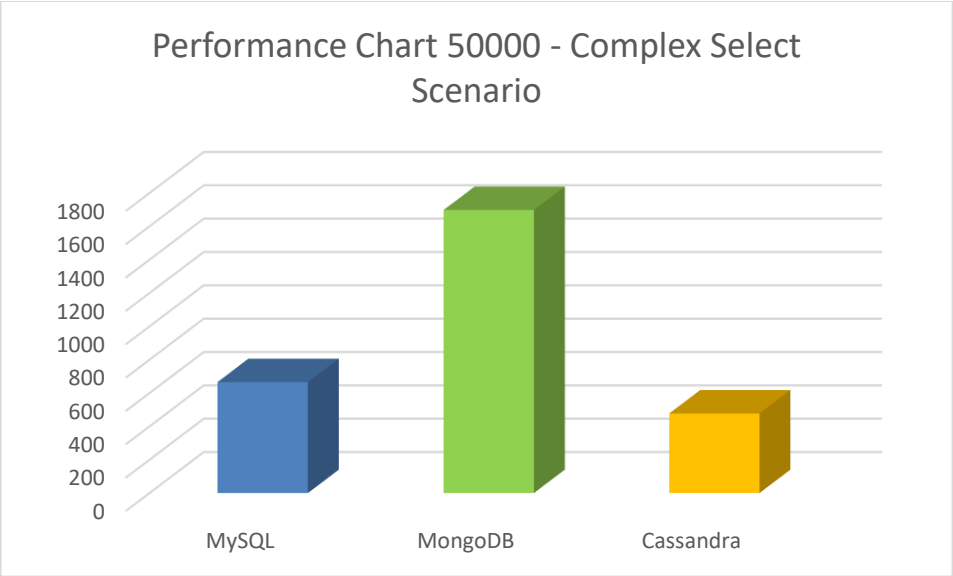


Figure 26 - Performance Chart - Complex Select Scenario - Volume 50000

7.2.2.6 Complex Select Scenario with Indexing

The indexing techniques were only applied to the complex select scenario to have clearer answers from the data volume impacts. However, it should be done for the other operations like Insert, Update and Delete, but due to the author's time constraints, it was not possible to be done.

The optimisations done in MySQL were the removal of the *GROUP BY* operation and adding *DISTINCT* instead, to remove the number of indexing operations. Also, a new index was created on the review table for the column *review_create_date_time* to optimise the *ORDER BY* operation.

For MongoDB, a single index was created in the collection *wifi_spot_visit* and the property *wifi_spot_id* to optimise the retrieval of visits for each Wi-Fi spot.

For Apache Cassandra, the available optimisations did not follow the author's intentions, so they were not done. Some of the available optimisations required changing the data model, like denormalising tables and this denormalisation was done initially when defining the data model in the section 5.2.3.3, there were also indexing strategies, but those were already applied in the implementation, explained in the section 6.3.3. Although the results of Apache Cassandra without optimisations were already positive.

In Table 25, there are the performance results for the complex select scenario with indexing and with a data volume of 50000.

Database	**Response Time (seconds)**					
	Executions (1 to 5)					**Average**
MySQL	474	408	553	597	408	488
MongoDB	5.58	5.65	5.9	4.38	5.07	5.316
Apache Cassandra	-	-	-	-	-	-

Table 25 - Performance results for the complex select scenario with indexing, with data volume 50000

The application of indexing techniques had a substantial impact on the performance of the complex select scenario, especially for MongoDB. With indexing enabled, MongoDB achieved an average response time of just 5.316 seconds, a dramatic improvement when compared to its previously recorded average of 1689 seconds for the same volume without indexing.

In contrast, MySQL, even after applying optimisations, achieved an average of 488 seconds. While this does reflect a significant improvement compared to the earlier unoptimised result of 645 seconds, it remains substantially slower than MongoDB's indexed performance.

It allows us to conclude that indexing strategies have a significant impact on the result, especially on MongoDB. In contrast, while MySQL also sees improvements through indexing, its relational structure and query execution mechanisms reveal performance bottlenecks when handling extensive datasets. Cassandra, even without indexing changes, demonstrated strong baseline performance, confirming its suitability for high-volume analytical queries in denormalised environments.

7.2.3 Energy

This section presents the energy results given by Kepler for the different scenarios and data volumes.

7.2.3.1 Insert Scenario

Table 26, Table 27 and Table 28 indicate the energy results for the insert scenario.

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	92.796	89.901	81.375	75.594	73.644	82.662
MongoDB	7.458	3.606	1.629	0.444	0.402	2.7078
Apache Cassandra	11.568	18.048	13.302	13.713	24.423	16.2108

Table 26 - Energy results for the insert scenario with data volume 500

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	716.646	600.657	638.946	704.913	714.882	675.2088
MongoDB	29.448	17.106	13.485	8.661	16.101	16.9602
Apache Cassandra	170.409	79.569	74.403	85.374	101.55	102.261

Table 27 - Energy results for the insert scenario with data volume 5000

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	6657.435	6664.908	6916.489	6712.046	6895.881	6769.352
MongoDB	293.493	338.643	228.675	251.397	251.832	272.808

Apache Cassandra	748.086	2044.92	912.182	958.026	949.314	1122.506

Table 28 - Energy results for the insert scenario with data volume 50000

The energy consumption analysis for the insert scenario reveals significant disparities in how efficiently each database handles write operations. MongoDB consistently consumed the least energy across all tested data volumes. At 500 records, it required only 2.71 joules on average, a stark contrast to MySQL's 82.66 joules and Cassandra's 16.21 joules. This efficiency was maintained at higher volumes: with 5000 records, MongoDB averaged just 16.96 joules, compared to 675.21 joules for MySQL and 102.26 joules for Cassandra. The contrast became even more pronounced at 50000 records, where MongoDB used 272.808 joules on average, while Apache Cassandra required 1122.506 joules, and MySQL consumed an extremely high 6769.352 joules.

From the graphical figures, Figure 27, Figure 28, and Figure 29, and the results mentioned above, it is possible to conclude that MongoDB is the most energy-efficient system for insert-heavy workloads, particularly as data volumes scale. MySQL, in contrast, demonstrated extremely high energy consumption at all volumes, pointing out that it demands more computational resources. Apache Cassandra, while significantly more efficient than MySQL, still required more energy than MongoDB, especially at higher volumes. Overall, MongoDB stands out as the most sustainable choice in terms of energy efficiency for large-scale insert scenarios.

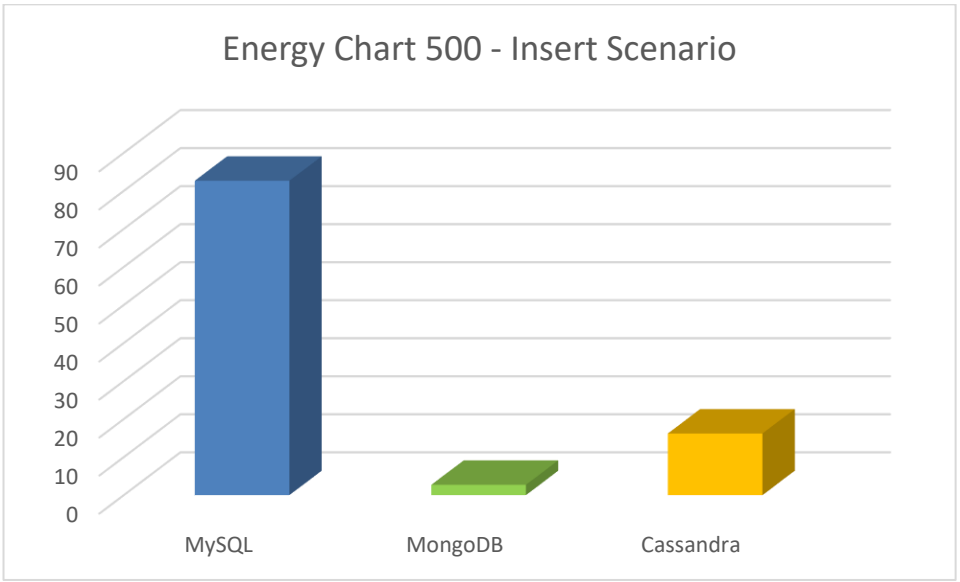


Figure 27 - Energy Chart - Insert Scenario - Volume 500

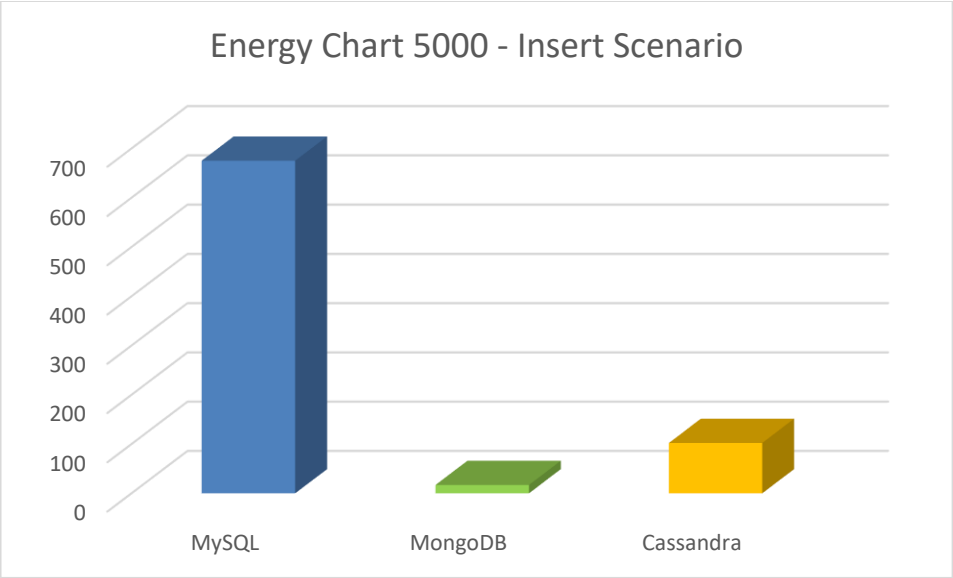


Figure 28 - Energy Chart - Insert Scenario - Volume 5000

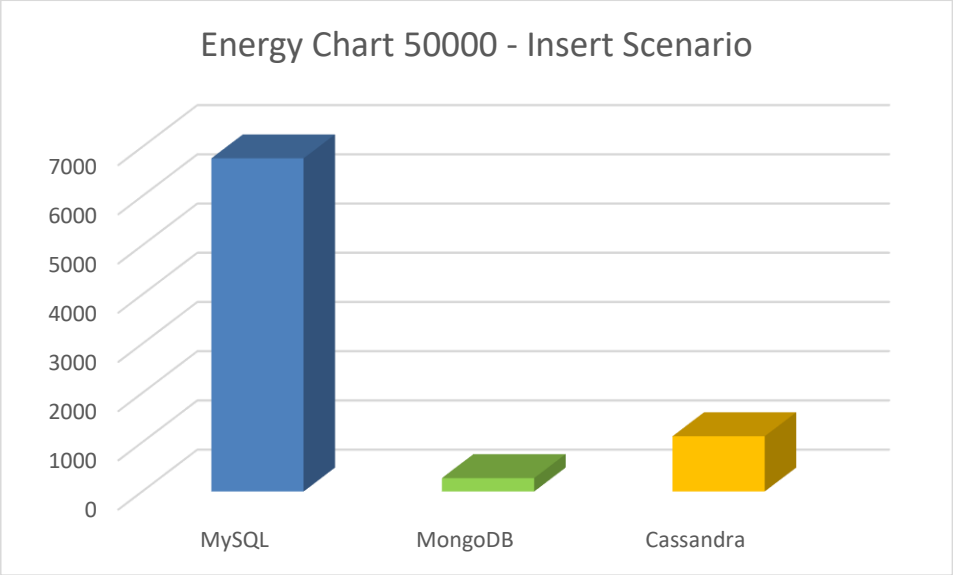


Figure 29 - Energy Chart - Insert Scenario - Volume 50000

7.2.3.2 Update Scenario

The energy results for the update scenario are indicated in Table 29, Table 30 and Table 31.

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	47.871	46.698	89.685	77.223	58.434	63.9822
MongoDB	12.099	7.062	7.032	9.237	9.48	8.982
Apache Cassandra	20.574	17.715	13.377	20.589	10.38	16.527

Table 29 - Energy results for the update scenario with data volume 500

Database	Response Time (Joules)

	Executions (1 to 5)					Average
MySQL	814.134	774.501	528.213	502.839	576.198	639.177
MongoDB	53.088	41.568	39.192	33.393	44.247	42.2976
Apache Cassandra	224.814	181.371	175.413	205.611	202.449	197.9316

Table 30 - Energy results for the update scenario with data volume 5000

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	2402.337	2406.94	2418.669	2450.416	2385.539	2412.78
MongoDB	385.752	420.434	339.915	380.749	392.307	383.8314
Apache Cassandra	1156.116	1427.745	1159.887	1164.957	1157.09	1213.159

Table 31 - Energy results for the update scenario with data volume 50000

The update scenario reveals clear differences in energy consumption between the databases. With a small dataset of 500 records, MongoDB was the most energy-efficient, averaging just 8.98 joules, followed by Apache Cassandra with 16.53 joules, and MySQL, which consumed the most at 63.98 joules. As data volume increased to 5000 records, MongoDB maintained its lead, averaging 42.30 joules, while Cassandra rose to 197.93 joules, and MySQL's consumption increased to 639.18 joules. At the highest tested volume of 50000 records, the disparity grew even more pronounced: MongoDB consumed 383.8314 joules, while Cassandra reached 1213.159 joules, and MySQL reached 2412.78 joules.

From Figure 30, Figure 31, and Figure 32, it is possible to conclude that these results demonstrate that MongoDB is consistently the most energy-efficient option for UPDATE operations across all tested data volumes. Apache Cassandra performs reasonably well, especially at medium volumes. In contrast, MySQL is the least energy-efficient system, with high and rapidly increasing energy usage.

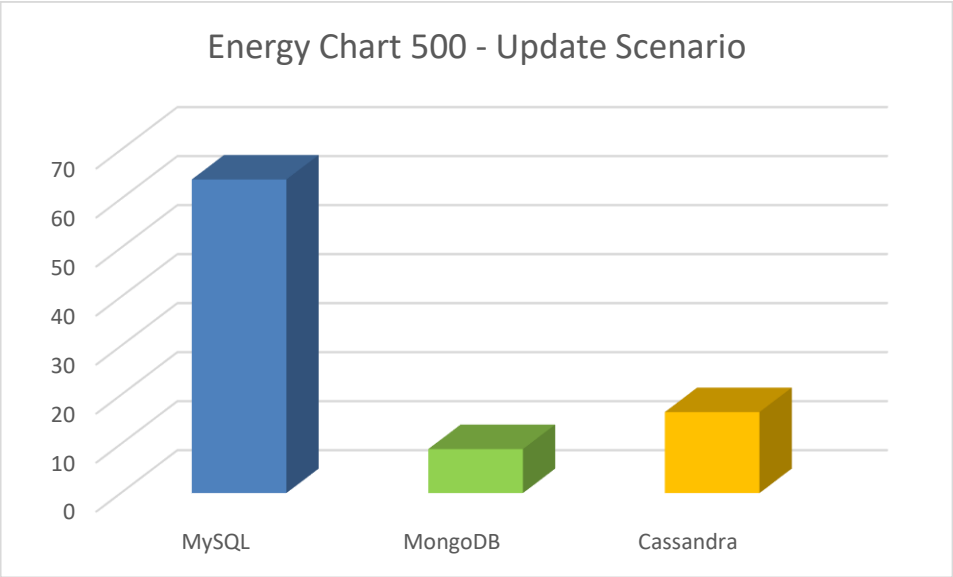


Figure 30 - Energy Chart - Update Scenario - Volume 500

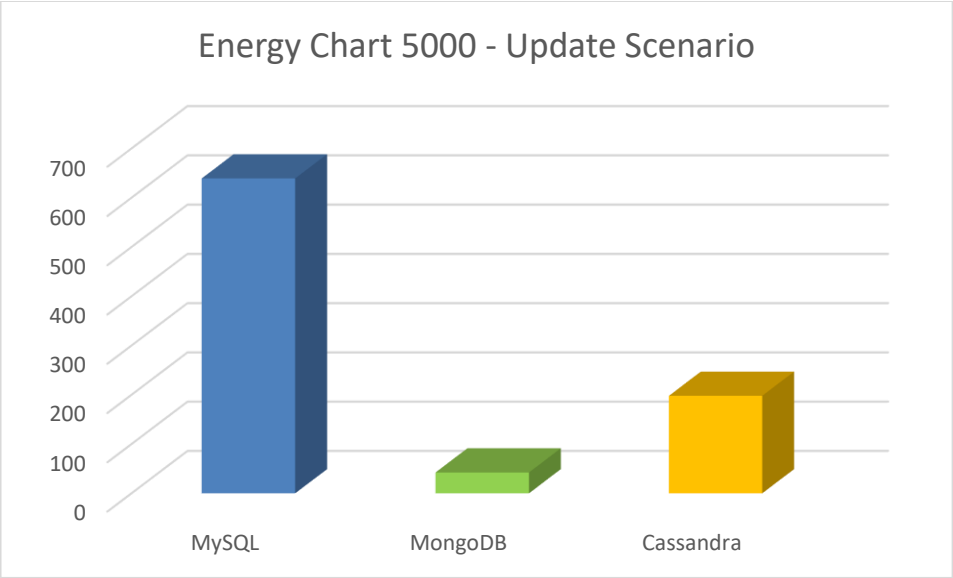


Figure 31 - Energy Chart - Update Scenario - Volume 5000

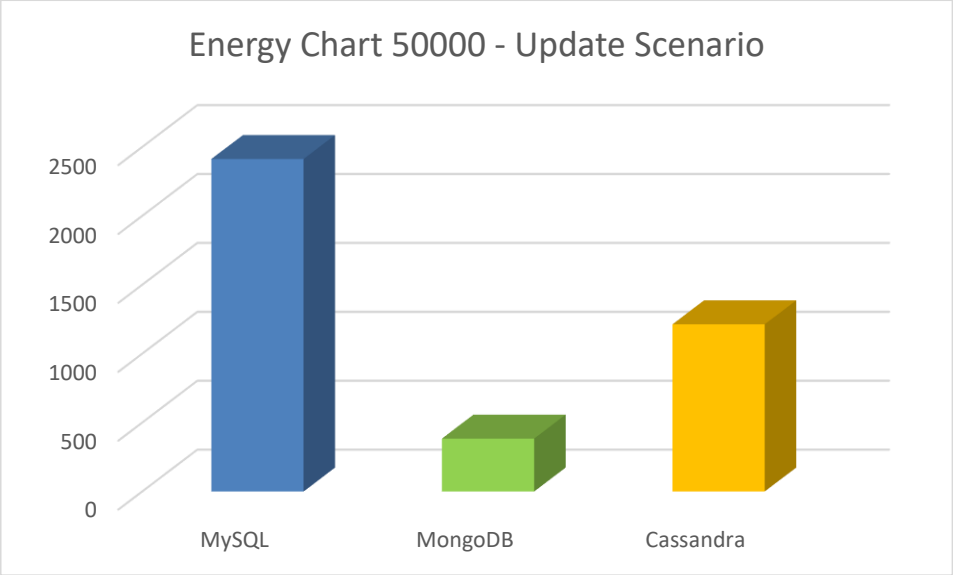


Figure 32 - Energy Chart - Update Scenario - Volume 50000

7.2.3.3 Delete Scenario

Table 32, Table 33 and Table 34 have the energy results measured for the delete scenario.

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	56.898	68.25	93.255	68.751	79.026	73.236
MongoDB	4.125	3.348	1.767	3.714	2.574	3.1056
Apache Cassandra	7.062	5.982	6.636	5.22	4.035	5.787

Table 32 - Energy results for the delete scenario with data volume 500

Database	Response Time (Joules)

	Executions (1 to 5)					Average
MySQL	651.024	785.82	693.206	712.764	750.168	718.5964
MongoDB	71.88	15.948	6.711	16.833	7.641	24.8026
Apache Cassandra	74.961	77.655	82.089	69.027	76.929	76.1322

Table 33 - Energy results for the delete scenario with data volume 5000

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	7275.549	7195.454	7539.897	7434.898	7592.579	7407.675
MongoDB	164.337	286.731	158.031	60.84	100.755	154.1388
Apache Cassandra	611.754	533.31	582.716	834.588	635.634	639.6004

Table 34 - Energy results for the delete scenario with data volume 50000

The delete scenario results show substantial differences in energy consumption across the evaluated databases, particularly as the volume of deletions increases. MongoDB consistently demonstrated the best energy efficiency. At 500 records, it consumed an average of just 3.11 joules per execution, whereas Apache Cassandra used nearly twice as much at 5.79 joules, and MySQL was considerably higher at 73.24 joules. As the data volume increased to 5,000 records, the energy gap widened significantly: MongoDB required only 24.80 joules on average, while Cassandra consumed 76.13 joules and MySQL soared to 718.60 joules. The disparity became even more pronounced at 50,000 records, where MongoDB maintained a relatively modest consumption of 154.1388 joules, Cassandra reached 639.6004 joules, and MySQL recorded an exceptionally high value of 7407.675 joules.

These findings establish MongoDB as the most energy-efficient solution for DELETE operations at all tested scales. It is suitable for applications where frequent deletion of data is required and power consumption is a critical concern. Apache Cassandra, while significantly more efficient than MySQL, still trails behind MongoDB, particularly at higher volumes. MySQL's extremely high energy usage highlights its inefficiency in handling DELETE operations under large-scale workloads. Overall, MongoDB offers a clear advantage in terms of energy savings for delete-intensive tasks, making it a strong candidate for sustainable database management in write-heavy systems. Figure 33, Figure 34, and Figure 35 allow us to understand these findings more easily. Like the performance study of this scenario, the data volumes do not influence each database’s behaviour.

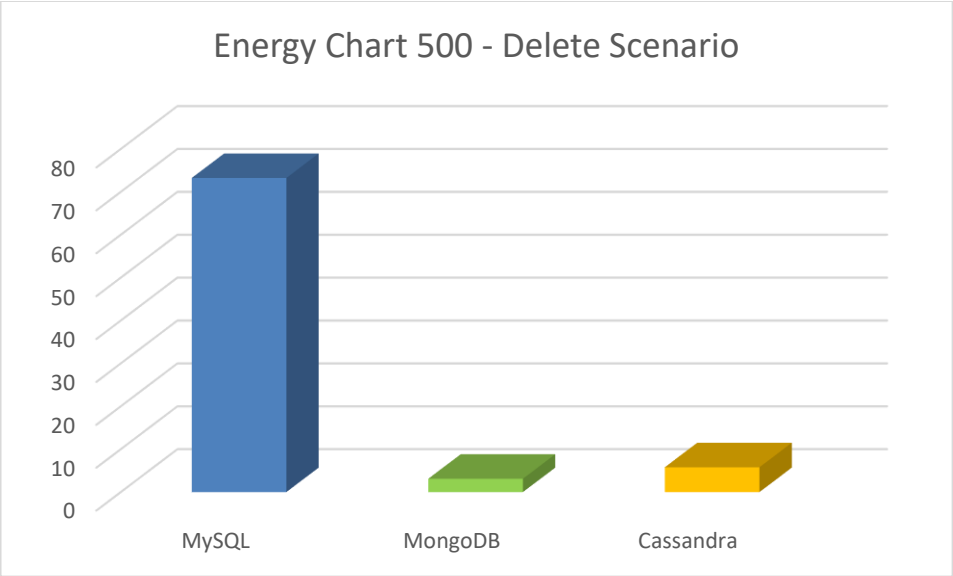


Figure 33 - Energy Chart - Delete Scenario - Volume 500

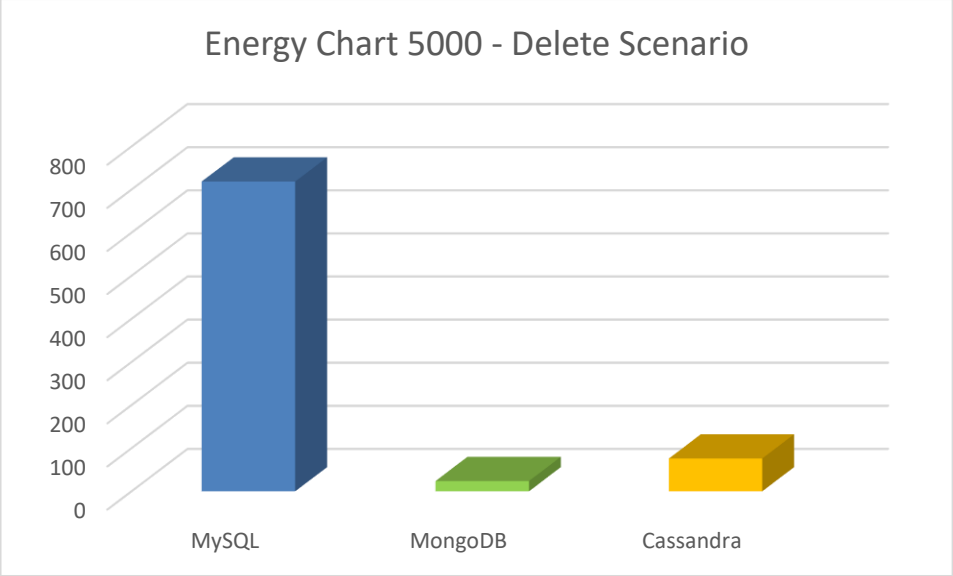


Figure 34 - Energy Chart - Delete Scenario - Volume 5000

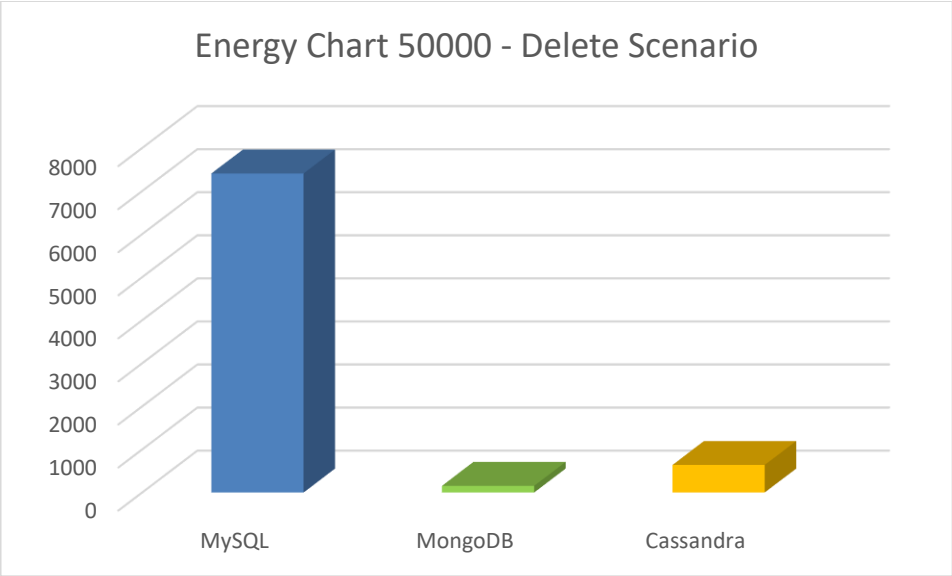


Figure 35 - Energy Chart - Delete Scenario - Volume 50000

7.2.3.4 Simple Select Scenario

The energy results for the simple select scenario are in Table 35, Table 36 and Table 37.

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	3	2.148	1.203	3.105	1.872	2.2656
MongoDB	0.747	0.276	0.489	0.837	0.411	0.552
Apache Cassandra	0.411	0.429	0.504	0.777	0.381	0.5004

Table 35 - Energy results for the simple select scenario with data volume 500

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	1.113	1.614	1.386	0.564	0.747	1.0848
MongoDB	0.609	0.381	0.905	0.687	0.444	0.6052
Apache Cassandra	0.657	0.672	0.078	0.231	1.068	0.5412

Table 36 - Energy results for the simple select scenario with data volume 5000

Database	Response Time (Joules)					
	Executions (1 to 5)					Average
MySQL	0.672	1.263	5.616	0.321	0.474	1.6692
MongoDB	0.321	0.078	0.672	0.063	0.231	0.273
Apache Cassandra	0.702	0.519	0.411	0.429	0.762	0.5646

Table 37 - Energy results for the simple select scenario with data volume 50000

The energy consumption results for the simple select scenario highlight MongoDB and Apache Cassandra as the most energy-efficient databases across all tested volumes. At 500 records, MongoDB required just 0.552 joules on average, closely followed by Cassandra at 0.500 joules, while MySQL consumed significantly more at 2.27 joules. This trend persisted with 5000 records,

where Cassandra slightly outperformed MongoDB with an average of 0.541 joules compared to 0.605 joules. MySQL remained the least efficient at this scale as well, averaging 1.08 joules. At the highest data volume of 50000 records, MongoDB once again demonstrated the best energy efficiency, consuming just 0.272 joules on average. Cassandra followed with 0.5646 joules, while MySQL settled at 1.6692 joules. All the results can be seen graphically in Figure 36, Figure 37, and Figure 38.

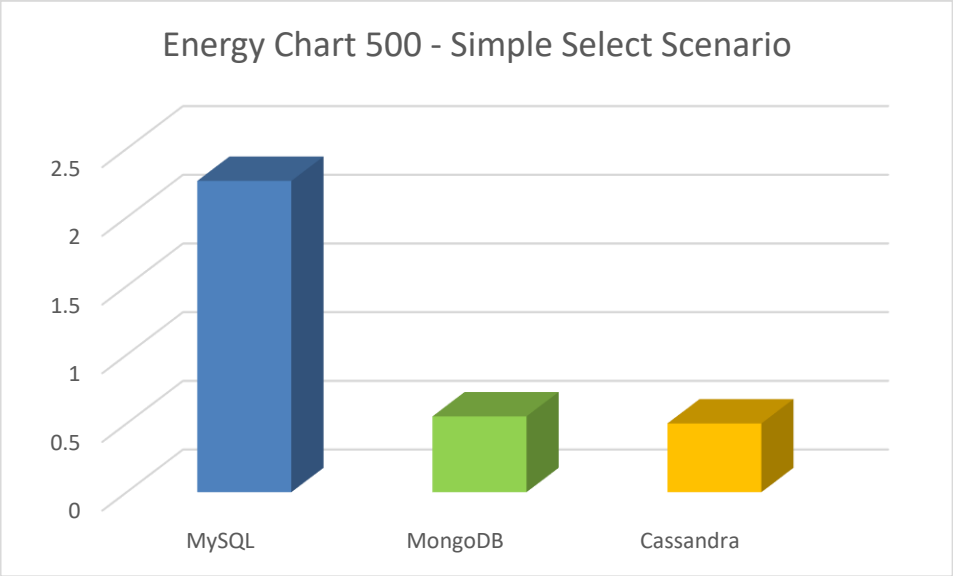


Figure 36 - Energy Chart - Simple Select Scenario - Volume 500

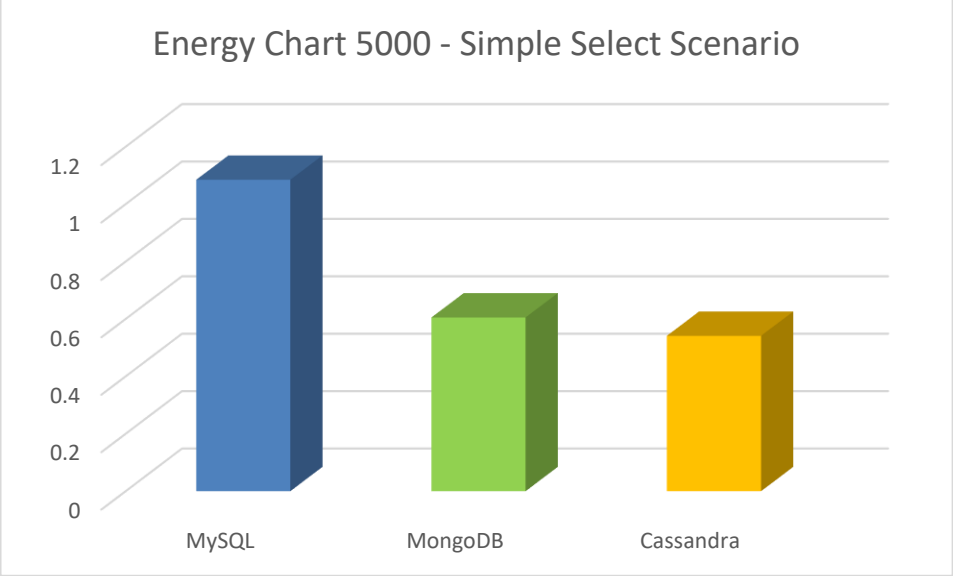


Figure 37 - Energy Chart - Simple Select Scenario - Volume 5000

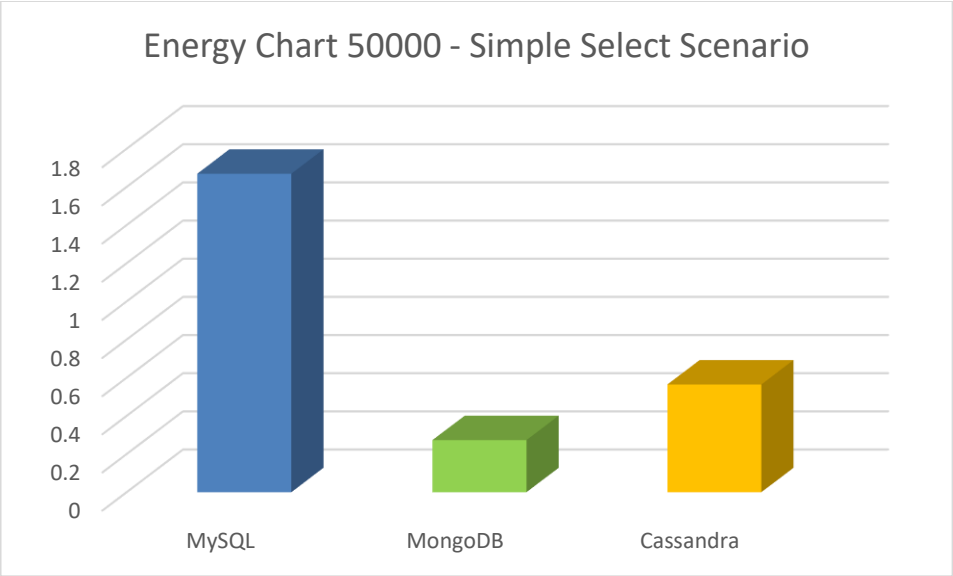


Figure 38 - Energy Chart - Simple Select Scenario - Volume 50000

These findings indicate that MongoDB consistently maintains low energy usage for simple read operations, making it the most energy-conscious option overall. Apache Cassandra also exhibits strong efficiency, particularly at mid-range volumes. In contrast, MySQL's higher energy consumption at lower volumes suggests less optimised query processing, although it shows improvement at scale. Overall, MongoDB stands out as the most energy-efficient choice for simple select scenarios, particularly when sustained low power consumption is a priority.

7.2.3.5 Complex Select Scenario

The complex select scenario has a lot of disparate results. Those results are seen in Table 38, Table 39 and Table 40.

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	42.804	62.859	37.053	60.534	49.578	50.5656
MongoDB	24.042	29.838	21.258	15.933	30.372	24,2886
Apache Cassandra	42.777	23.088	24.231	30.648	17.532	27.6552

Table 38 - Energy results for the complex select scenario with data volume 500

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	911.55	633.42	512.724	518.649	546.999	624.6684
MongoDB	19.224	11.97	16.164	13.284	12.405	14.6094
Apache Cassandra	334.137	204.249	170.388	174.408	147.288	206.094

Table 39 - Energy results for the complex select scenario with data volume 5000

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	5599.128	5872.593	5748.462	5616.806	5719.311	5711.26

MongoDB	83.331	97.517	82.769	94.365	85.729	88.7422
Apache Cassandra	1288.803	1431.186	1353.174	1281.684	1438.02	1358.573

Table 40 - Energy results for the complex select scenario with data volume 50000

The complex select scenario revealed substantial differences in energy consumption between the evaluated databases, particularly as data volume increased. MongoDB consistently demonstrated superior energy efficiency, beginning with an average of 24.29 joules at 500 records, significantly lower than MySQL's 50.57 joules and Apache Cassandra's 27.66 joules. This trend continued at the 5000-record scale, where MongoDB required only 14.61 joules on average, while Cassandra's consumption increased to 206.09 joules and MySQL to 624.67 joules. At the highest data volume of 50000 records, the differences became even more pronounced: MongoDB maintained a relatively low consumption of 88.7422 joules, in contrast to Cassandra's 1358.573 joules and MySQL's extremely high 5711.26 joules.

These results establish MongoDB as the most energy-efficient solution for handling complex read operations, especially as data scales. Its' consistently low energy usage across all volumes suggests highly optimised query execution. Apache Cassandra also shows reasonably good energy performance, particularly in smaller workloads, but its energy demands increase significantly with larger datasets. MySQL, while functional, proved to be the least energy-efficient option, consuming dramatically more power as query complexity and data volume grew. Overall, MongoDB is the most sustainable choice for complex queries where energy efficiency is a primary consideration. In Figure 39, Figure 40, and Figure 41, these conclusions can be graphically seen where the disparity of MongoDB is clear.

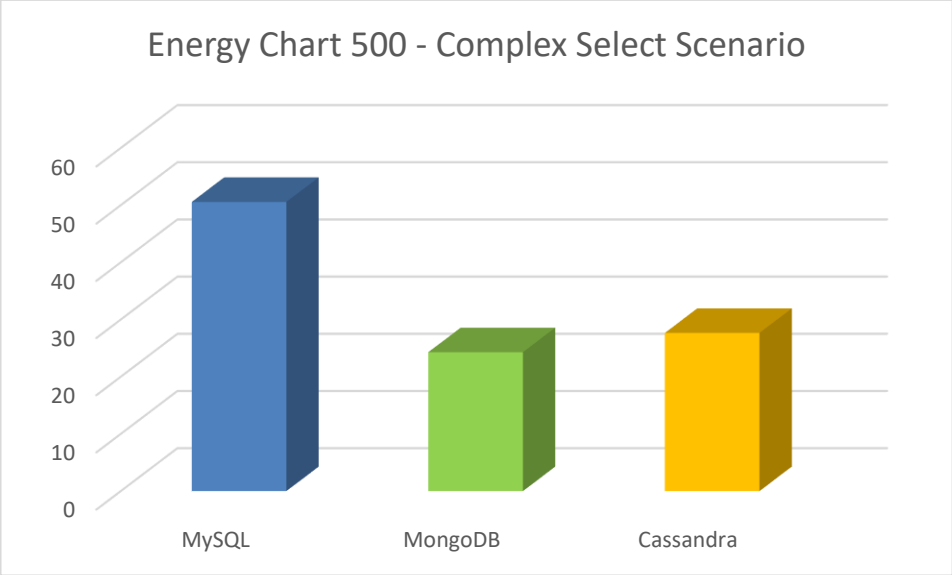


Figure 39 - Energy Chart - Complex Select Scenario - Volume 500

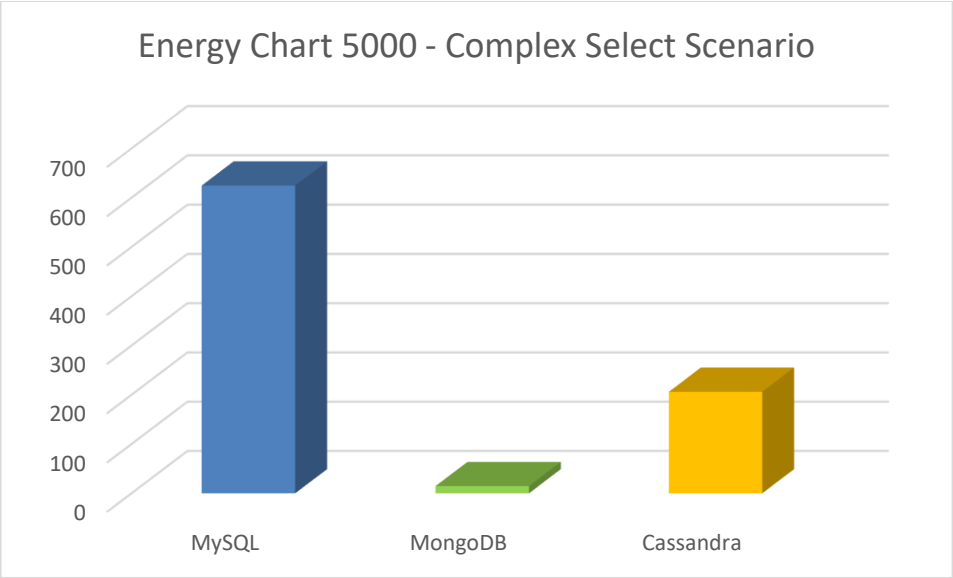


Figure 40 - Energy Chart - Complex Select Scenario - Volume 5000

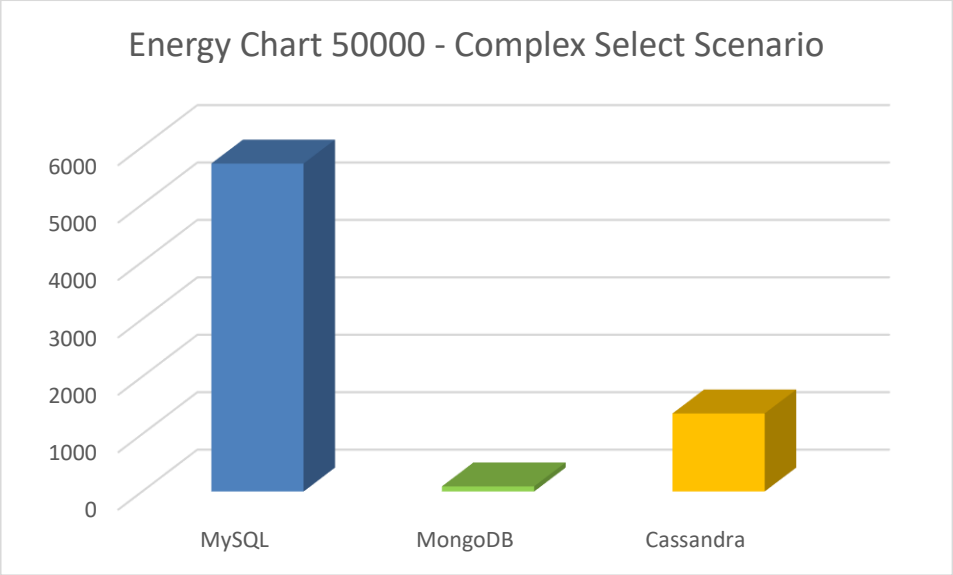


Figure 41 - Energy Chart - Complex Select Scenario - Volume 50000

7.2.3.6 Complex Select Scenario with Indexing

Table 41 shows the energy consumption results for the complex select scenario with indexing following the same optimisations mentioned in the section 7.2.2.6.

Database	**Response Time (Joules)**					
	Executions (1 to 5)					**Average**
MySQL	3961.029	3793.986	4423.305	4740.066	3420.591	4067.795
MongoDB	37.374	13.951	14.715	17.778	13.638	19.4892
Apache Cassandra	-	-	-	-	-	-

Table 41 - Energy results for the complex select scenario with indexing with data volume 50000

The energy consumption results for the complex select scenario with indexing reveal a dramatic difference between MongoDB and MySQL, demonstrating the significant impact of targeted indexing strategies. MongoDB's energy usage dropped sharply to an average of 19.49 joules. This is a reduction of over 75% compared to its unindexed value of 88.74 joules for the same scenario and data volume.

In stark contrast, MySQL, despite optimisations, exhibited extremely high energy consumption. The average value of 4067.80 joules significantly reduced from its earlier unindexed performance of 5711.26 joules, but it has a notorious discrepancy compared with MongoDB, showing little benefit from the applied indexing in terms of energy efficiency.

Even without optimisations, Apache Cassandra's earlier results demonstrate much more moderate energy consumption than MySQL, though not as low as MongoDB.

Overall, these findings confirm that MongoDB not only achieves exceptional performance gains with indexing but also maintains superior energy efficiency, even under complex workloads. MySQL, on the other hand, continues to show substantial energy consumption, highlighting the limitations of its engine and execution strategy under large-scale. Indexing in MongoDB proves to be a highly effective strategy both for reducing execution time and conserving energy.

7.3 Hypothesis Tests

To enable more robust interpretation of the experimental results, hypothesis tests were conducted for each scenario using one-way ANOVA [82]. One-way ANOVA hypothesis testing is a statistical method used to test whether the means of multiple groups are equal [82].

The p-value is computed as the probability that a test statistic falls in this extreme region, given the null hypothesis. The test then rejects H0 if this generalised p-value is less than a chosen significance level [82].

The significance level considered was 95% (p-value = 0.05). The null hypothesis (H0) assumes no significant difference between the database systems (MySQL, MongoDB, and Apache Cassandra), while the alternative hypothesis (H1) asserts that at least two differ significantly. Rejections or acceptances are based on the obtained p-values.

To obtain the p-values for each scenario and each data volume, a Python script was developed using the library *scipy* [83]. This library receives the five executions and calculates the p-value. The Python script is visible in the GitHub repository [84].

Table 42 and Table 43 gather all the p-values calculated for each scenario, including insert, update, delete, simple and complex select scenarios and across the different data volumes for performance and energy, respectively.

Scenario	Data Volume		
	500	5000	50000
Insert	0.0000000	0.0000000	0.0000000
Update	0.0000005	0.0897412	0.0000000
Delete	0.0000015	0.0000000	0.0000000
Simple Select	0.0001305	0.0001693	0.7556707
Complex Select	0.0003040	0.2978730	0.0000000

Table 42 - ANOVA Tests and p-values calculated for each scenario and data volume related to performance results

Scenario	Data Volume		
	500	5000	50000
Insert	0.0000000	0.0000000	0.0000000
Update	0.0000098	0.0000003	0.0000000
Delete	0.0000000	0.0000000	0.0000000
Simple Select	0.0001169	0.0667719	0.2419545
Complex Select	0.0013272	0.0000032	0.0000000

Table 43 - ANOVA Tests and p-values calculated for each scenario and data volume related to energy results

This way, with all the p-values calculated, it is possible to do the hypothesis testing for each scenario and data volume:

- Insert Scenario
    - Performance
        - **H0:** There is no significant difference in the average insert response time between MySQL, MongoDB, and Apache Cassandra.
        - **H1:** There is a significant difference in the average insert response time between at least two of the databases.
        - **Response:** Reject H0. ANOVA tests returned p-values of 0.0000000 for all volumes (500, 5000, and 50000), all well below the 0.05 threshold. This indicates statistically significant differences in insert performance. MongoDB consistently showed significantly lower response times.
    - Energy
        - **H0:** There is no significant difference in energy consumption for insert operations between the databases.
        - **H1:** There is a significant difference in energy consumption for insert operations between at least two databases.
        - **Response:** Reject H0. All p-values were 0.0000000, confirming statistically significant differences. MongoDB consumed significantly less energy than MySQL and Cassandra across all volumes.
- Update Scenario

- Performance
  - **H0:** There is no significant difference in average update performance among MySQL, MongoDB, and Apache Cassandra.
  - **H1:** At least one database differs significantly in update performance.
  - **Response:** Reject H0. ANOVA returned p-values of 0.0000005 (volume 500), 0.0897412 (volume 5000), and 0.0000000 (volume 50000). Although volume 5000's p-value exceeds 0.05, the remaining results indicate statistical significance at other volumes. MongoDB performed best at lower volumes, while Cassandra was strongest at volume 50000.
- Energy
  - **H0:** There is no significant difference in energy consumption during update operations across the three databases.
  - **H1:** At least one database differs significantly in energy consumption for update operations.
  - **Response:** Reject H0. ANOVA tests returned p-values of 0.0000098 (volume 500), 0.0000003 (volume 5000), and 0.0000000 (volume 50000), all below 0.05, indicating statistically significant energy consumption differences. MongoDB was the most energy-efficient at all volumes.

- Delete Scenario
  - Performance
    - **H0:** There is no significant difference in the average delete response time across the three databases.
    - **H1:** There is a significant difference in the average delete response time between at least two databases.
    - **Response:** Reject H0. P-values were 0.0000015 (volume 500) and 0.0000000 (volumes 5000 and 50000), confirming statistically significant performance differences. MongoDB deleted records significantly faster than MySQL and Cassandra.
  - Energy
    - **H0:** There is no significant difference in energy consumption during delete operations between the databases.
    - **H1:** There is a significant difference in energy consumption during delete operations between at least two databases.
    - **Response:** Reject H0. All volumes returned p-values of 0.0000000, confirming strong evidence of significant energy differences. MongoDB was the most efficient in all cases.
- Simple Select Scenario
  - Performance
    - **H0:** There is no significant difference in the average response time for simple select queries across the three databases.
    - **H1:** There is a significant difference in the average response time for simple select queries between at least two databases.

        - **Response:** Reject H0 for volumes 500 (p = 0.0001305) and 5000 (p = 0.0001693), but do not reject H0 for volume 50000 (p = 0.7556707). This suggests that performance differences exist only for lower data volumes, being MongoDB the best.
    - Energy
        - **H0:** There is no significant difference in energy consumption for simple select queries between the databases.
        - **H1:** There is a significant difference in energy consumption for simple select queries between at least two databases.
        - **Response:** Reject H0 for volume 500 (p = 0.0001169), but do not reject H0 for volumes 5000 (p = 0.0667719) and 50000 (p = 0.2419545). Statistically significant energy differences only exist at low volumes with Apache Cassandra being more energy efficient.
- Complex Select Scenario
    - Performance
        - **H0:** There is no significant difference in the average response time for complex select queries between the databases.
        - **H1:** There is a significant difference in the average response time for complex select queries between at least two databases.
        - **Response:** Reject H0 for volumes 500 (p = 0.0003040) and 50000 (p = 0.0000000), but do not reject H0 at volume 5000 (p = 0.2978730). MongoDB performed well at lower volumes but degraded sharply with scale, while MySQL and Cassandra were more consistent at high volumes.
    - Energy
        - **H0:** There is no significant difference in energy consumption for complex select queries between the databases.
        - **H1:** There is a significant difference in energy consumption for complex select queries between at least two databases.
        - **Response:** Reject H0. All p-values were below 0.05: 0.0013272 (volume 500), 0.0000032 (volume 5000), and 0.0000000 (volume 50000). MongoDB was significantly more energy-efficient than the others, especially under large loads.

7.4 Performance and Energy Tradeoffs

This section summarises and correlates the performance and energy consumption results for each scenario, achieving the Goal of the Goal Question Metric defined in the section 7.1.

At this point, and to achieve the Goal, it is already possible to answer all four questions of the GQM:

- **Q1: What is the performance of each operation in relational databases?**

In relational databases, MySQL, for INSERT operations, the response time increased with data volume: 40.32 seconds at 500 records, 209.6 seconds at 5,000 records, and 1999 seconds at 50,000 records. In UPDATE operations, the response times were 5.63 seconds at 500 records, 81 seconds at 5,000, and 561.4 seconds at 50,000. DELETE operations also showed progressive degradation: from 22.35 seconds at 500 records to 245.2 seconds at 5,000 and 2456.6 seconds at 50,000. In contrast, simple select queries maintained stable and low latency values: 0.044 seconds for 500 records, 0.054 seconds for 5,000, and 0.039 seconds for 50,000. Complex select queries showed significantly longer execution times: 5.81 seconds at 500 records, 67.8 seconds at 5,000, and 665.2 seconds at 50,000.

- **Q2: What is the performance of each operation in non-relational databases?**
  In INSERT operations, MongoDB consistently showed the best response times across all volumes: 0.17 seconds at 500 records, 2.86 seconds at 5,000, and 35.146 seconds at 50,000. Apache Cassandra required 4.95 seconds, 24.67 seconds, and 276.4 seconds for the same volumes, respectively.
  For UPDATE operations, MongoDB recorded averages of 0.63 seconds at 500 records, 61.84 seconds at 5,000, and 1664.8 seconds at 50,000. Apache Cassandra showed 3.79 seconds, 67.8 seconds, and 412.2 seconds, respectively.
  In DELETE operations, MongoDB showed response times of 0.20 seconds at 500 records, 2.94 seconds at 5,000, and 25.076 seconds at 50,000. Apache Cassandra recorded 1.57 seconds, 23.18 seconds, and 203.2 seconds for the same volumes.
  Simple select queries in MongoDB yielded 0.016 seconds (500 records), 0.022 seconds (5,000), and 0.040 seconds (50,000), while Cassandra reported 0.031 seconds, 0.021 seconds, and 0.050 seconds, respectively.
  In the complex select scenario, MongoDB presented average response times of 2.02 seconds at 500 records, 59.76 seconds at 5,000, and 1697.6 seconds at 50,000. Apache Cassandra returned 7.28 seconds, 63.18 seconds, and 477.2 seconds for the same volumes.
- **Q3: What is the energy consumption of each operation in relational databases?**
  In relational databases, MySQL, in INSERT operations, energy consumption rose consistently with data volume: 82.66 J at 500 records, 675.21 J at 5,000 records, and 6668.35 J at 50,000 records. UPDATE operations followed a similar trend: 63.98 J, 639.18 J, and 2412.78 J, respectively. For DELETE operations, the results were 73.24 J, 718.60 J, and 7407.68 J across the three volumes. Energy use for simple selects remained comparatively low, averaging 2.27 J at 500 records, 1.08 J at 5,000, and 1.67 J at 50,000. However, complex selects led to much higher consumption: 50.57 J for 500 records, 624.67 J for 5,000, and 5711.26 J for 50,000.
- **Q4: What is the energy consumption of each operation in non-relational databases?**
  For INSERT operations, MongoDB showed the lowest energy footprint: 2.71 J at 500 records, 16.96 J at 5,000, and 272.80 J at 50,000. Cassandra consumed 16.21 J, 102.26 J, and 1122.506 J for the same volumes.
  UPDATE operations followed the same order: MongoDB used 8.98 J, 42.30 J, and 383.83 J; Cassandra consumed 16.53 J, 197.93 J, and 1213.16 J. In DELETE scenarios, MongoDB

averaged 3.11 J at 500 records, 24.80 J at 5,000, and 154.14 J at 50,000, while Cassandra required 5.79 J, 76.13 J, and 639.60 J.

Simple select queries were extremely energy-efficient in MongoDB (0.55 J, 0.61 J, and 0.27 J) and Cassandra (0.50 J, 0.54 J, and 0.56 J). Complex select operations showed more pronounced differences: MongoDB consumed 24.29 J (500 records), 14.61 J (5,000), and 88.74 J (50,000), while Cassandra reported 27.66 J, 206.09 J, and 1358.57 J, respectively.

In these experiments, the error rate is not a significant metric to be considered, since it was not used in concurrency, and the number of executions was not massive. However, the error rate was 0% across all scenarios and operations. This result, although not indicative of robustness under stress or concurrency, suggests that the implementations used in each database were functionally correct under the tested conditions. The absence of errors indicates that the basic CRUD operations were performed as expected and that the test scripts, database drivers, and data models were correctly integrated. While this does not guarantee stability at scale, it provides a level of confidence in the correctness of the experimental setup and in the ability of each system to handle typical requests without failure in isolated, low-concurrency environments.

Having the questions answered, with the insights and quick conclusions in sections 7.2.2 and 7.2.3, and with the hypothesis tests done in section 7.3, it is possible to achieve the goal. In sections 7.4.1, 7.4.2, 7.4.3, 7.4.4, and 7.4.5, it is defined the **"Tradeoffs between Performance and Energy Consumption for relational and non-relational databases for each operation**".

7.4.1 Insert Scenario

In examining the insert scenario, the relationship between performance and energy consumption across MySQL, MongoDB, and Apache Cassandra reveals distinct systemic tradeoffs beyond mere speed or efficiency in isolation. A critical observation is that for MySQL, performance degradation is tightly coupled with a proportional increase in energy consumption. As INSERT operations became slower with increasing data volumes, energy usage surged in parallel. This pattern implies that MySQL does not scale well in environments where inserts dominate.

MongoDB, by contrast, showed a different behaviour. Its performance remained high even at larger volumes, and crucially, its energy usage stayed proportionally low. This consistent efficiency suggests that MongoDB benefits from a lightweight, document-based model that minimises processing complexity per INSERT operation. Thus, MongoDB not only performs well but does so without energy penalties, in contrast to what section 1.2 describes that increased performance often leads to increased energy consumption. This can be due to the versions of MySQL and MongoDB used, or the drivers used for the connections, since MySQL used JDBC and MongoDB used the MongoDB Java Driver.

Apache Cassandra showed an intermediate profile. While its insert performance was generally lower than MongoDB's, its energy consumption did not rise as sharply as MySQL's. However, as

the data volume grew, the divergence between response time and energy efficiency became more pronounced. Cassandra, while not immediately reflected in insert time, manifests an increased energy usage, especially at the 50,000 record scale. This indicates that although Cassandra scales better than MySQL in raw performance, its energy cost per insert becomes less predictable and less efficient as workloads intensify.

Overall, systems like MySQL that prioritise strong transactional guarantees may suffer from performance-energy coupling under load, making them inefficient in high-volume environments. MongoDB's decoupling of performance and energy usage makes it exceptionally well-suited for scalability.

7.4.2 Update Scenario

The update scenario reveals contrasting behaviours across the databases in how performance correlates with energy efficiency. MySQL's performance correlates with its energy consumption because when the volume of data increases, its energy and performance also increase. This indicates that its architecture struggles to maintain efficiency under growing write pressure, making it the least suitable for update-heavy operations.

In contrast, MongoDB begins as the fastest and most energy-efficient option at small volumes, but its performance deteriorates quickly as data size increases. Interestingly, even though MongoDB's execution time became the worst at 50,000 records, its energy consumption remained lower than the other systems. This decoupling suggests that MongoDB, despite longer execution times under stress, still uses system resources more efficiently.

Apache Cassandra doesn't lead in either speed or energy efficiency at low volumes, but it scales more predictably. As update volume increases, its energy use rises, yet at a slower rate than MySQL. It's a distributed, eventually consistent model that appears to handle larger workloads with more stable resource demands, despite modest inefficiencies compared to MongoDB.

In summary, MySQL exhibits a tightly coupled decline in both performance and energy efficiency. MongoDB decouples energy use from execution time, maintaining low consumption even under performance degradation. Cassandra offers the most balanced tradeoff, scaling more gracefully than MySQL while being more energy-intensive than MongoDB.

7.4.3 Delete Scenario

In the delete scenario, a strong correlation emerges between performance degradation and energy consumption across systems, most clearly in MySQL. As delete times increased dramatically with data volume, energy usage escalated in parallel, indicating poor scalability. At 50,000 deletions, MySQL's energy usage reached a critical level, confirming that its tightly coupled transactional model struggles under high deletion loads, both in latency and power consumption.

MongoDB exhibited the most efficient behaviour. It maintained extremely low delete times at all scales, and crucially, its energy consumption remained low even as volume grew. This decoupling of performance and energy under stress suggests that MongoDB handles deletions with minimal overhead, making it highly suitable for write-heavy applications with frequent data pruning.

Apache Cassandra scaled better than MySQL in both metrics, but didn't match MongoDB's efficiency. As data volume increased, Cassandra's performance degraded and energy consumption increased, but at a more stable rate compared to MySQL.

MongoDB consistently achieves fast deletions with minimal energy usage, even at scale. Cassandra offers a more stable tradeoff, while MySQL becomes increasingly inefficient. These tradeoffs make MongoDB the strongest candidate for scalable and energy-conscious deletion-heavy systems.

7.4.4 Select Scenarios

In the simple select scenario, all databases achieved low latency, with only minor variations in execution time. MongoDB led slightly in both performance and energy efficiency, but Apache Cassandra followed closely, and even MySQL maintained competitive speeds. Energy consumption, however, introduced clearer distinctions. MongoDB consistently consumed the least energy, followed by Cassandra, while MySQL lagged with significantly higher energy usage, especially at low volumes. This suggests that even when raw performance appears similar, MongoDB processes simple queries with lower system overhead.

In contrast, the complex select scenario introduced significant performance divergence. MongoDB, while initially the fastest and most energy-efficient, saw its performance degrade sharply with increasing data volumes, like the studies mentioned in the section 4.5. At the largest scale, it became the slowest system, although it still consumed the least energy overall. Conversely, Apache Cassandra, which was the slowest in the simple scenario, scaled more gracefully under complexity, outperforming MongoDB in latency at high volumes but consuming more energy. MySQL, while neither the fastest nor most efficient, demonstrated steady performance growth with complexity, thanks to its support for relational JOIN operations.

Overall, the contrast between the two scenarios highlights that query complexity impacts not just execution time but also how efficiently each system uses hardware resources. MongoDB is the most energy-conscious across both scenarios, even when performance degrades. Cassandra improves its relative performance under complex load, but at a growing energy cost. MySQL shows a predictable yet inefficient pattern; its performance remains reasonable, but energy usage escalates with complexity and volume.

7.4.5 Indexing Scenario

The introduction of indexing in the complex select scenario significantly altered performance and energy profiles, especially for MongoDB. Its execution time dropped from 1697.6 seconds to just over 5 seconds, and energy consumption fell by more than 75%, demonstrating that MongoDB scales exceptionally well when indexing is applied.

MySQL also improved with indexing, but the gains were comparatively modest. Execution time decreased, and energy usage remained extremely high. This suggests that while indexing helps, MySQL's underlying architecture imposes limitations that are not easily resolved through optimisation alone, which goes against what was studied and concluded in the literature review in section 4.5.

Apache Cassandra, although not further optimised, maintained consistent performance and energy efficiency. Its earlier results already positioned it as a reliable option for complex queries at scale, owing to its denormalised model.

In summary, indexing strategies adopted in this experiment have a transformative effect for complex queries in MongoDB, a moderate one in MySQL, and limited relevance for Cassandra. The impact of indexing on write operations (e.g., INSERT, UPDATE, DELETE) was not addressed in this study. However, indexing may degrade performance in such cases due to the overhead of maintaining index structures.

8 Conclusion

This chapter summarises the study's achievements and contributions, difficulties encountered, existing threats that may have affected the validity of the work, future work and final considerations.

8.1 Achievements and contributions

The main goal of this study was to explore how different types of databases, relational and non-relational, handle the tradeoffs between performance and energy use, especially as query complexity and data volume grow. By carefully testing common operations like insert, update, delete, and select across various dataset sizes, we were able to see how the underlying architecture of each database affects its efficiency and resource usage. This helped clarify how scaling impacts both energy consumption and performance in real-world scenarios.

These experiments proved that performance relates to energy consumption because, taking MySQL as an example, its energy consumption increased while its performance increased too, even with small data volumes, which goes against what was concluded in the section 4.5. On the other hand, MongoDB maintained low energy usage despite performance drops, while Cassandra provided a middle-ground solution with stable performance and moderate energy demands.

Another interesting finding came from experiments involving indexing during more complex queries. MongoDB responded especially well to small indexing tweaks, with a noticeable drop in execution time and energy consumption. MySQL, however, showed more modest improvements, limited in part by the fixed structure of its relational model.

8.2 Difficulties

One of the most pressing obstacles was the limited computational resources available for conducting large-scale experiments. Even at a relatively modest volume of 50,000 records, execution times were considerably high, severely constraining the time left to complete the remaining phases of the research.

These challenges underline the multifaceted nature of benchmarking modern database systems, where performance depends not only on the workload but also on tuning, resource isolation, and architecture-specific behaviours.

8.3 Threats to validity

The validity of the findings is subject to a few considerations. First, the tests were executed in a controlled environment with all systems deployed on the same physical machine, which has limited resources for heavy experiments. While this enabled consistent measurement, it does not fully reflect distributed production setups.

Moreover, the testing scenarios, although diverse, were limited to one type of application context and workload pattern. Different schemas, query shapes, or deployment scales may influence the results.

Additional indexing strategies across all workloads could have influenced the overall balance of performance and energy use because it was only applied to selected scenarios due to time restrictions. The author's lack of experience with non-relational databases may have affected these optimisation studies and consequently led to less accurate conclusions about this subject.

8.4 Future work

Multiple promising avenues can be pursued in future research. One potential path involves expanding the scope of indexing evaluations to encompass write-intensive operations, such as updates, deletions, and insertions, to assess how these optimisations affect energy consumption and performance in scenarios that go beyond predominantly read-focused workloads.

Several data models could be adopted to leverage the advantages of each database. For example, normalising the data more in MongoDB and Apache Cassandra could improve the results of updates. So, in future work, the adoption of several data models may be considered to expand the experimental conclusions. Also, other databases should be considered because, as concluded in the section 7.2.2.5, non-relational databases have different behaviours among them, including different types of non-relational databases like key-value based ones.

Simulating environments under real-world conditions would include accounting for factors like network delays, data replication strategies, and system failover behaviours to explore how these variables impact overall energy efficiency. Future studies could also enhance the evaluation framework by including additional performance indicators, such as request throughput and detailed latency distributions, for a better understanding of system behaviour.

The experiments used only one virtual user. In the future, it would be important to consider more virtual users because real-world applications typically involve concurrent access, which can significantly impact performance and energy consumption due to increased system load, contention, and resource utilisation.

8.5 Final considerations

There are several promising directions for future research. One potential path is to expand the indexing experiments by including write-intensive operations, such as inserts, updates, and deletions, to better understand how these optimisations affect energy use and performance in workloads that aren't primarily read-focused. Another idea is to adopt more complex, domain-specific data models and test a wider variety of database systems, including key-value stores, to gain a more comprehensive view.

Experiments with real-world scenarios would involve accounting for factors like network latency, data replication methods, and system failover mechanisms, which could all influence energy efficiency in practical scenarios. Additionally, future work could improve the testing environment by incorporating more detailed performance metrics, such as request throughput and full latency distributions, to better capture how systems behave under different loads.

References

[1] R. Čerešňák and M. Kvet, “Comparison of query performance in relational a non-relation databases,” *Transportation Research Procedia*, vol. 40, pp. 170–177, Jan. 2019, doi: 10.1016/J.TRPRO.2019.07.027.

[2] S. Schmid, W. Reinhardt, and E. Galicz, “3 NoSQL-DB’s for geoapplications,” Jun. 2015.

[3] P. Ciancarini *et al.*, “Analysis of Energy Consumption of Software Development Process Entities,” *Electronics 2020, Vol. 9, Page 1678*, vol. 9, no. 10, p. 1678, Oct. 2020, doi: 10.3390/ELECTRONICS9101678.

[4] M. Shah, A. Kothari, and S. Patel, “A Comprehensive Survey on Energy Consumption Analysis for NoSQL,” *Scalable Computing: Practice and Experience*, vol. 23, no. 1, pp. 35–50, Apr. 2022, doi: 10.12694/SCPE.V23I1.1971.

[5] A. Bonvoisin, C. Quinton, and R. Rouvoy, “Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks,” p. 11, Mar. 2024, doi: 10.13039/501100001665.

[6] K. Fraczek and M. Plechawska-Wojcik, “Comparative Analysis of Relational and Non-relational Databases in the Context of Performance in Web Applications,” *Communications in Computer and Information Science*, vol. 716, pp. 153–164, 2017, doi: 10.1007/978-3-319-58274-0_13.

[7] R. Pereira *et al.*, “Energy efficiency across programming languages: How do energy, time, and memory relate?,” *SLE 2017 - Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2017*, pp. 256–267, Oct. 2017, doi: 10.1145/3136014.3136031.

[8] “Most used languages among software developers globally 2024| Statista.” Accessed: May 17, 2025. [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[9] M. Altman, A. Cohen, and K. Nissim, “ACM TechBrief: Data Privacy Protection,” *ACM TechBrief: Data Privacy Protection*, Jul. 2024, doi: 10.1145/3679004.

[10] D. Gotterbarn, K. Miller, and S. Rogerson, “Software engineering code of ethics,” *Commun ACM*, vol. 40, no. 11, 1997, doi: 10.1145/265684.265699.

[11] “Code of Ethics | National Society of Professional Engineers.” Accessed: Nov. 24, 2024. [Online]. Available: https://www.nspe.org/resources/ethics/code-ethics

[12] “Diário da República, 2.ª série PARTE E Artigo 2.º,” Nov. 2020.

[13] D. Gotterbarn, K. Miller, and S. Rogerson, “Software engineering code of ethics,” *Commun ACM*, vol. 40, no. 11, 1997, doi: 10.1145/265684.265699.

[14] D. Ganesh Chandra, “BASE analysis of NoSQL database,” *Future Generation Computer Systems*, vol. 52, pp. 13–21, Nov. 2015, doi: 10.1016/J.FUTURE.2015.05.003.

[15] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, “A Survey and Comparison of Relational and Non-Relational Database,” Aug. 2012. [Online]. Available: www.ijert.org

[16] “memcached - a distributed memory object caching system.” Accessed: May 04, 2025. [Online]. Available: https://memcached.org/

[17] “Redis - The Real-time Data Platform.” Accessed: May 04, 2025. [Online]. Available: https://redis.io/

[18] “Bigtable: Fast, Flexible NoSQL | Google Cloud.” Accessed: May 04, 2025. [Online]. Available: https://cloud.google.com/bigtable

[19] “Apache Cassandra | Apache Cassandra Documentation.” Accessed: May 04, 2025. [Online]. Available: https://cassandra.apache.org/_/index.html

[20] “MongoDB: The World’s Leading Modern Database | MongoDB.” Accessed: May 04, 2025. [Online]. Available: https://www.mongodb.com/

[21] “Couchbase: Best Free NoSQL Cloud Database Platform.” Accessed: May 04, 2025. [Online]. Available: https://www.couchbase.com/

[22] “Elasticsearch as a NoSQL Database | Elastic Blog.” Accessed: Feb. 08, 2025. [Online]. Available: https://www.elastic.co/blog/found-elasticsearch-as-nosql

[23] Mark. Richards, *Software architecture patterns*. O’Reilly Media, 2015.

[24] P. Marco and M. Marcu, “Impact of the persistence layer on performance and architecture for a Java web application,” *SAMI 2022 - IEEE 20th Jubilee World Symposium on Applied Machine Intelligence and Informatics, Proceedings*, pp. 261–266, 2022, doi: 10.1109/SAMI54271.2022.9780780.

[25] Pratik. Patel and Karl. Moss, *Java database programming with JDBC*. Coriolis Group Books, 1997.

[26] “Using Transactions (The Java™ Tutorials > JDBC Database Access > JDBC Basics).” Accessed: Nov. 02, 2024. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

[27] N. Dhingra, “Analysis of ORM Based JPA Implementations,” 2017, doi: 10.20381/RUOR-20804.

[28] G. Gajos, “Hibernate: The Silver Bullet | Toptal®.” Accessed: Oct. 13, 2024. [Online]. Available: https://www.toptal.com/java/how-hibernate-ruined-my-career

[29] A. Thomasian, “Performance Analysis of Database Systems,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1769, pp. 305–327, 2000, doi: 10.1007/3-540-46506-5_13.

[30] T. Kanij, R. Merkel, and J. Grundy, “Performance assessment metrics for software testers,” *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering, CHASE 2012 - Proceedings*, pp. 63–65, 2012, doi: 10.1109/CHASE.2012.6223025.

[31] M. Premal, B. Nirpal, and K. V Kale, “A Brief Overview Of Software Testing Metrics,” *International Journal on Computer Science and Engineering (IJCSE)*, Jan. 2011.

[32] G. Diprima, “Comparison of tools and metrics for non-functional testing of web pages,” *2022 | Number*, p. 58, 2022, Accessed: Dec. 27, 2024. [Online]. Available: http://www.theseus.fi/handle/10024/784211

[33] Apache Software Foundation, “Apache JMeter - Apache JMeter™.” Accessed: Dec. 28, 2024. [Online]. Available: https://jmeter.apache.org/

[34] Grafana Labs, “Load testing for engineering teams | Grafana k6.” Accessed: Dec. 28, 2024. [Online]. Available: https://k6.io/

[35] J. Kunkel and M. F. Dolz, “Understanding hardware and software metrics with respect to power consumption,” *Sustainable Computing: Informatics and Systems*, vol. 17, pp. 43–54, Mar. 2018, doi: 10.1016/J.SUSCOM.2017.10.016.

[36] G. M. Dat *et al.*, “Metrics of energy consumption in software systems: a systematic literature review,” *IOP Conf Ser Earth Environ Sci*, vol. 431, no. 1, p. 012051, Feb. 2020, doi: 10.1088/1755-1315/431/1/012051.

[37] L. Papadopoulos, A. Ampatzoglou, C. Marantos, A. Chatzigeorgiou, G. Digkas, and D. Soudris, “Interrelations between software quality metrics, performance and energy consumption in embedded applications,” *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES 2018*, pp. 62–65, May 2018, doi: 10.1145/3207719.3207736.

[38] A. Chatzigeorgiou and G. Stephanides, “Energy Metric for Software Systems,” *Software Quality Journal*, vol. 10, no. 4, pp. 355–371, 2002, doi: 10.1023/A:1022142105380/METRICS.

[39] T. Leonhard, J. Peslalz, and B. Katz, “Experimental Analysis of Optimization Techniques for Energy Efficiency in Distributed Systems,” 2024, Accessed: Dec. 27, 2024. [Online]. Available: https://opus4.kobv.de/opus4-hm/frontdoor/index/index/docId/573

[40] Cloud Native Computing Foundation, “Monitoring Container Power Consumption with Kepler - Kepler.” Accessed: Dec. 27, 2024. [Online]. Available: https://sustainable-computing.io/design/metrics/?utm_source=chatgpt.com

[41] M. Amaral, T. Eilam, H. Chen, E. Kyung Lee, and S. Choochotkaew, “Exploring Kepler’s potentials: unveiling cloud application power consumption | CNCF.” Accessed: Dec. 27, 2024. [Online]. Available: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/?utm_source=chatgpt.com

[42] F. Kifetew, D. Prandi, and A. Susi, “On the Energy Consumption of Test Generation”, Accessed: Apr. 19, 2025. [Online]. Available: https://www.kernel.org/doc/html/latest/power/powercap/powercap.html

[43] A. Noureddine, “PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools,” *2022 18th International Conference on Intelligent Environments, IE 2022 - Proceedings*, 2022, doi: 10.1109/IE54923.2022.9826760.

[44] M. J. Page *et al.*, “The PRISMA 2020 statement: an updated guideline for reporting systematic reviews,” *BMJ*, vol. 372, Mar. 2021, doi: 10.1136/BMJ.N71.

[45] M. Kuhrmann, D. M. Fernández, and M. Daneva, “On the pragmatic design of literature studies in software engineering: an experience-based guideline,” *Empir Softw Eng*, vol. 22, no. 6, pp. 2852–2891, Dec. 2017, doi: 10.1007/S10664-016-9492-Y/TABLES/8.

[46] A. T. Kabakus and R. Kara, “A performance evaluation of in-memory databases,” *Journal of King Saud University - Computer and Information Sciences*, vol. 29, no. 4, pp. 520–525, Oct. 2017, doi: 10.1016/J.JKSUCI.2016.06.007.

[47] G. Kiraz and C. Toğay, “IoT Data Storage: Relational & Non-Relational Database Management Systems Performance Comparison,” 2017.

[48] A. M. Bonteanu and C. Tudose, “Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA,” *Applied Sciences 2024, Vol. 14, Page 2743*, vol. 14, no. 7, p. 2743, Mar. 2024, doi: 10.3390/APP14072743.

[49] A.-M. Bonteanu, C. Tudose, and A. M. Anghel, “Performance Analysis for CRUD Operations in Relational Databases from Java Programs Using Hibernate,” in *2023 24th International Conference on Control Systems and Computer Science (CSCS)*, May 2023, pp. 655–662. doi: 10.1109/CSCS59211.2023.00109.

[50] M. Sharma, V. D. Sharma, and M. M. Bundele, “Performance Analysis of RDBMS and No SQL Databases: PostgreSQL, MongoDB and Neo4j,” in *2018 3rd International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*, Nov. 2018, pp. 1–5. doi: 10.1109/ICRAIE.2018.8710439.

[51] K. Kolonko, “Performance comparison of the most popular relational and non-relational database management systems,” 2018, Accessed: Nov. 30, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-16112

[52] B. Bani, Y.-G. Guéhéneuc, and F. Khomh, “Title: Understanding the Impact of Databases on the Energy Efficiency of Cloud Applications Directeurs de recherche,” 2016.

[53] W. Puangsaijai and S. Puntheeranurak, “A comparative study of relational database and key-value database for big data applications,” *2017 International Electrical Engineering Congress, iEECON 2017*, Oct. 2017, doi: 10.1109/IEECON.2017.8075813.

[54] L. S. Gubala Hari Babu and S. N. S. Dodla, “Comparative Analysis of Oracle and MySQL Databases : A Study on Query Execution and Scalability,” 2024, Accessed: Nov. 30, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-26813

[55] S. Reetishwaree and V. Hurbungs, “Evaluating the performance of SQL and NoSQL databases in an IoT environment,” *2020 3rd International Conference on Emerging Trends in Electrical, Electronic and Communications Engineering, ELECOM 2020 - Proceedings*, pp. 229–234, Nov. 2020, doi: 10.1109/ELECOM49001.2020.9297028.

[56] C. A. Győrödi, D. V. Dumşe-Burescu, D. R. Zmaranda, R. Győrödi, G. A. Gabor, and G. D. Pecherle, “Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application’s Data Storage,” *Applied Sciences 2020, Vol. 10, Page 8524*, vol. 10, no. 23, p. 8524, Nov. 2020, doi: 10.3390/APP10238524.

[57] P. Seda, J. Hosek, P. Masek, and J. Pokorny, “Performance testing of NoSQL and RDBMS for storing big data in e-applications,” in *2018 3rd International Conference on Intelligent Green Building and Smart Grid (IGBSG)*, Apr. 2018, pp. 1–4. doi: 10.1109/IGBSG.2018.8393559.

[58] H. S. Lella, R. Chattaraj, S. Chimalakonda, and M. Kurra, “Towards Comprehending Energy Consumption of Database Management Systems - A Tool and Empirical Study,” in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, in EASE ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 272–281. doi: 10.1145/3661167.3661174.

[59] “DB-Engines Ranking - popularity ranking of database management systems.” Accessed: Feb. 08, 2025. [Online]. Available: https://db-engines.com/en/ranking

[60] “Bulk Loading | Apache Cassandra Documentation.” Accessed: Mar. 23, 2025. [Online]. Available: https://cassandra.apache.org/doc/4.1/cassandra/operating/bulk_loading.html

[61] “How to Bulk Upload CSV file data into MySql Table? A very fast way using LOAD DATA. | by Kaustubh Joshi | Medium.” Accessed: Mar. 23, 2025. [Online]. Available: https://elpidaguy.medium.com/how-to-bulk-upload-csv-file-data-into-mysql-table-very-fast-way-using-load-data-statement-e3685890a568

[62] “JSON To MongoDB | MongoDB.” Accessed: Mar. 23, 2025. [Online]. Available: https://www.mongodb.com/resources/languages/json-to-mongodb

[63] “apache/jmeter: Apache JMeter open-source load testing tool for analyzing and measuring the performance of a variety of services.” Accessed: Jun. 15, 2025. [Online]. Available: https://github.com/apache/jmeter

[64] “grafana/k6: A modern load testing tool, using Go and JavaScript - https://k6.io.” Accessed: Jun. 15, 2025. [Online]. Available: https://github.com/grafana/k6

[65] “sustainable-computing-io/kepler: Kepler (Kubernetes-based Efficient Power Level Exporter) uses eBPF to probe performance counters and other system stats, use ML models to estimate workload energy consumption based on these stats, and exports them as Prometheus metrics.” Accessed: Jun. 15, 2025. [Online]. Available: https://github.com/sustainable-computing-io/kepler

[66] “joular/joularjx: JoularJX is a Java-based agent for software power monitoring at the source code level.” Accessed: Jun. 15, 2025. [Online]. Available: https://github.com/joular/joularjx

[67] “Java™ SE Development Kit 23, 23.0.1 Release Notes.” Accessed: Mar. 22, 2025. [Online]. Available: https://www.oracle.com/java/technologies/javase/23-0-1-relnotes.html

[68] E. Silva, R. Fidalgo, M. Ferro, and N. Franco, “Visual query languages to design complex queries: a systematic literature review,” *Softw Syst Model*, vol. 22, no. 4, pp. 1217–1249, Aug. 2023, doi: 10.1007/S10270-022-01071-4/TABLES/7.

[69] “How to Set Up and Configure MySQL in Docker | DataCamp.” Accessed: Apr. 19, 2025. [Online]. Available: https://www.datacamp.com/tutorial/set-up-and-configure-mysql-in-docker

[70] “Docker & MongoDB | Containers & Compatibility | MongoDB.” Accessed: Apr. 19, 2025. [Online]. Available: https://www.mongodb.com/resources/products/compatibilities/docker

[71] “Setting up Cassandra with Docker on Your Local Machine: A Step-by-Step Guide. | by Shruti Ghoradkar | Medium.” Accessed: Apr. 19, 2025. [Online]. Available: https://medium.com/@shrutighoradkar101/setting-up-cassandra-with-docker-on-your-local-machine-a-step-by-step-guide-40cdc1a41359

[72] “DIMEI_1191017/netquest-backend/docker-compose.yml at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/docker-compose.yml

[73] “DIMEI_1191017/netquest-backend/docker-compose-mongodb.yml at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/docker-compose-mongodb.yml

[74] “DIMEI_1191017/netquest-backend/docker-compose-cassandra.yml at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/docker-compose-cassandra.yml

[75] “Faker · PyPI.” Accessed: Apr. 19, 2025. [Online]. Available: https://pypi.org/project/Faker/

[76] “DIMEI_1191017/netquest-backend/scripts/mysql/mysql_script.py at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/scripts/mysql/mysql_script.py

[77] “DIMEI_1191017/netquest-backend/scripts/mongodb/mongodb_script.py at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/scripts/mongodb/mongodb_script.py

[78] “DIMEI_1191017/netquest-backend/scripts/cassandra/cassandra_script.py at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/scripts/cassandra/cassandra_script.py

[79] “DIMEI_1191017/netquest-backend/src/main/java/com/netquest/MongoDBConfig.java at main · 1191017/DIMEI_1191017.” Accessed: May 17, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/src/main/java/com/netquest/MongoDBConfig.java

[80] H. Koziolek, “Goal, Question, Metric,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4909 LNCS, pp. 39–42, 2008, doi: 10.1007/978-3-540-68947-8_6.

[81] V. R. Basili, G. Caldiera, and H. D. Rombach, “THE GOAL QUESTION METRIC APPROACH”.

[82] J. Gamage and S. Weerahandi, “Size performance of some tests in one-way anova,” *Commun Stat Simul Comput*, vol. 27, no. 3, pp. 625–640, 1998, doi: 10.1080/03610919808813500.

[83] “f_oneway — SciPy v1.15.3 Manual.” Accessed: Jun. 15, 2025. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f_oneway.html

[84] “DIMEI_1191017/netquest-backend/scripts/anova/tests.py at main · 1191017/DIMEI_1191017.” Accessed: Jun. 15, 2025. [Online]. Available: https://github.com/1191017/DIMEI_1191017/blob/main/netquest-backend/scripts/anova/tests.py

[85] “DIMEI_1191017/Documentação/Domain Model.svg at main · 1191017/DIMEI_1191017.” Accessed: Jun. 15, 2025. [Online]. Available:

https://github.com/1191017/DIMEI_1191017/blob/main/Documenta%C3%A7%C3%A3o/Domain%20Model.svg

[86] “Install k6 | Grafana k6 documentation.” Accessed: Apr. 19, 2025. [Online]. Available: https://grafana.com/docs/k6/latest/set-up/install-k6/

[87] “Load testing for engineering teams | Grafana k6.” Accessed: Apr. 19, 2025. [Online]. Available: https://k6.io/

Appendix A

This appendix shows the timeline of the Gantt Project that was developed to plan the dissertation thesis resolution. The timeline is presented in Figure 42 and Figure 43.

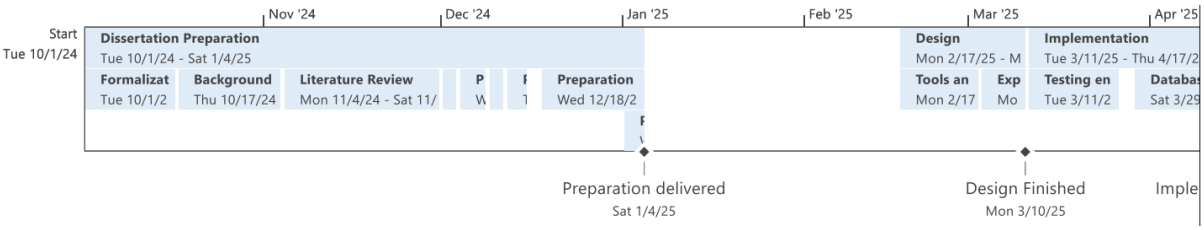


Figure 42 - Gantt Project Timeline - Part 1

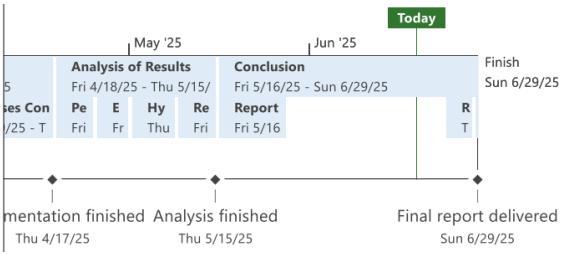
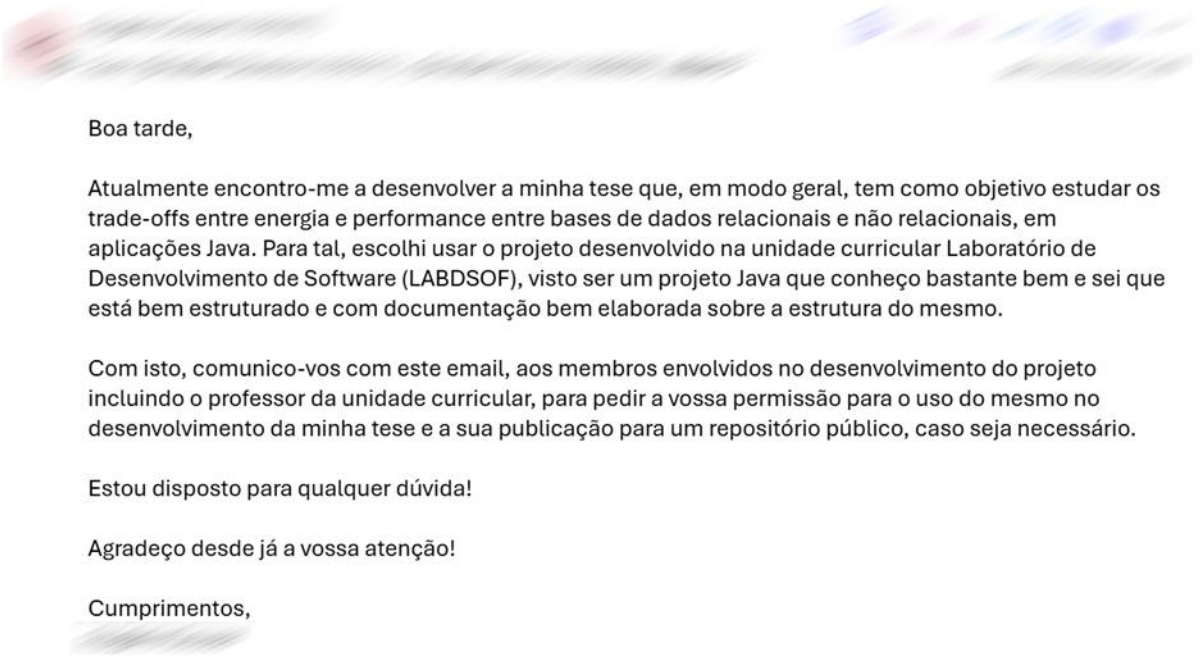


Figure 43 - Gantt Project Timeline - Part 2

Appendix B

This appendix shows the permissions given by all the group members and the teacher mentioned by the author to reuse the Java project developed during the Master's Degree. The images below (Figure 44, Figure 45, Figure 46, Figure 47, Figure 48, and Figure 49) show the email request and each answer.



Boa tarde,

Atualmente encontro-me a desenvolver a minha tese que, em modo geral, tem como objetivo estudar os trade-offs entre energia e performance entre bases de dados relacionais e não relacionais, em aplicações Java. Para tal, escolhi usar o projeto desenvolvido na unidade curricular Laboratório de Desenvolvimento de Software (LABDSOF), visto ser um projeto Java que conheço bastante bem e sei que está bem estruturado e com documentação bem elaborada sobre a estrutura do mesmo.

Com isto, comunico-vos com este email, aos membros envolvidos no desenvolvimento do projeto incluindo o professor da unidade curricular, para pedir a vossa permissão para o uso do mesmo no desenvolvimento da minha tese e a sua publicação para um repositório público, caso seja necessário.

Estou disposto para qualquer dúvida!

Agradeço desde já a vossa atenção!

Cumprimentos,

Figure 44 - Project usage email request sent

Bom dia
Não tenho objeção

Figure 45 - Project use approval from Teacher

Boa tarde,

Da minha parte tudo bem.

Cumprimentos,

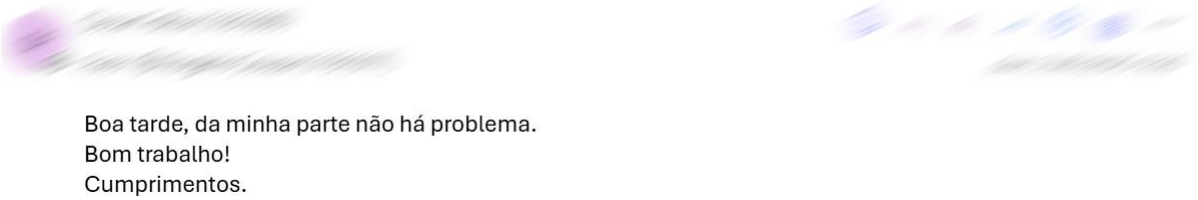
Figure 46 - Project use approval from Student 1

Boa tarde,

Da minha parte tudo bem.

Cumprimentos

Figure 47 - Project use approval from Student 2



Boa tarde, da minha parte não há problema.
Bom trabalho!
Cumprimentos.

Figure 48 - Project use approval from Student 3

Bom dia,
Do meu lado sem problemas.

Figure 49 - Project use approval from Student 4

Appendix C

This appendix contains the project's domain model separated by aggregates because it is too large to be presented in the document. The complete domain model can be seen in the project documentation [85].

Figure 50 represents the Review aggregate containing the necessary information to create one.

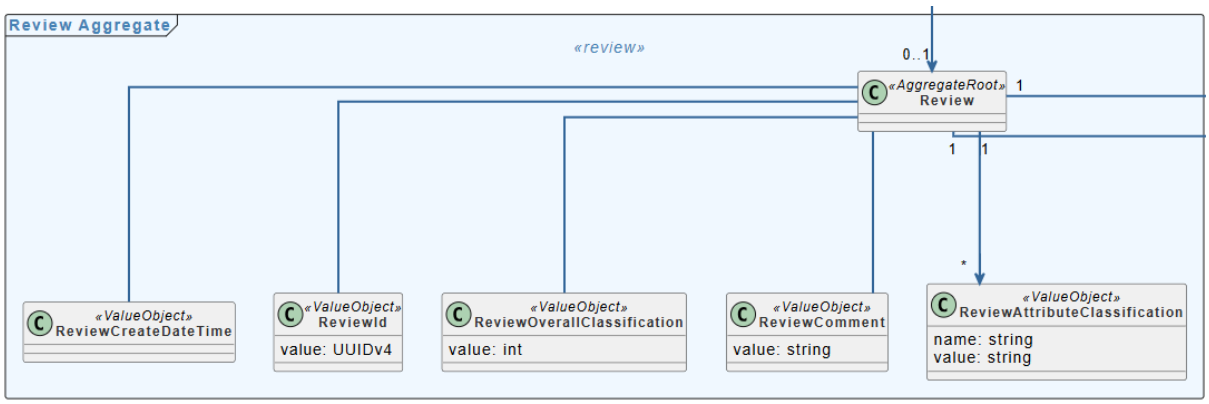


Figure 50 - Domain Model - Review Aggregate

Figure 51 is the Wifi Spot Visit Aggregate responsible for handling all the Wi-Fi spot visits of the project.

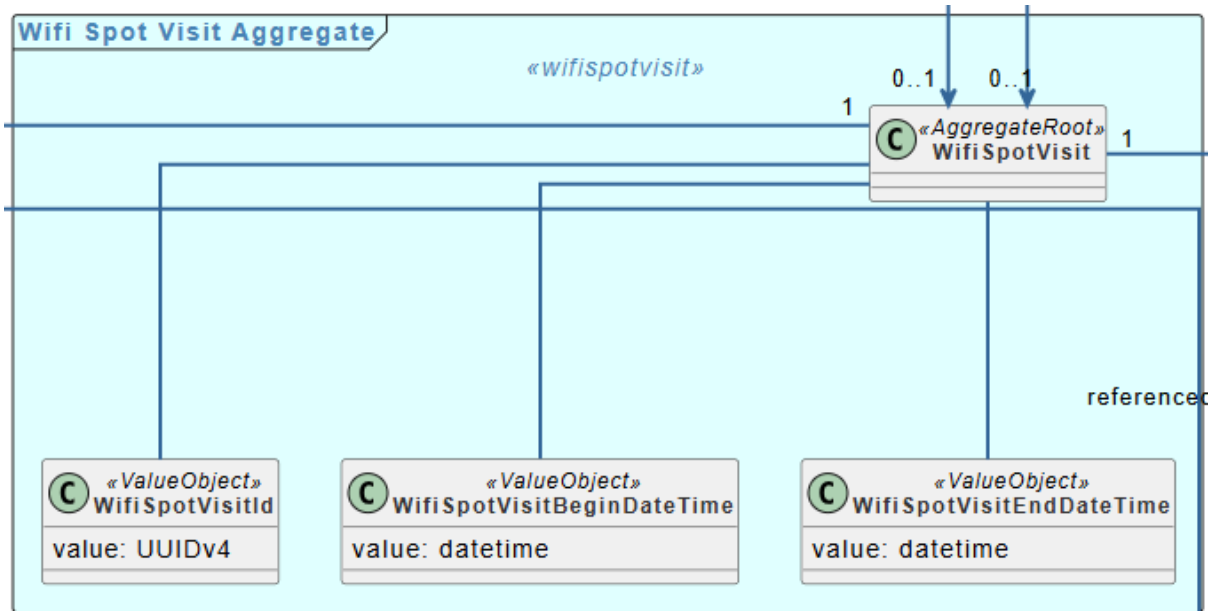


Figure 51 - Domain Model - Wifi Spot Visit Aggregate

The User Aggregate is shown on Figure 52 and it is responsible for dealing with the user's definition of the application, including the user's roles.

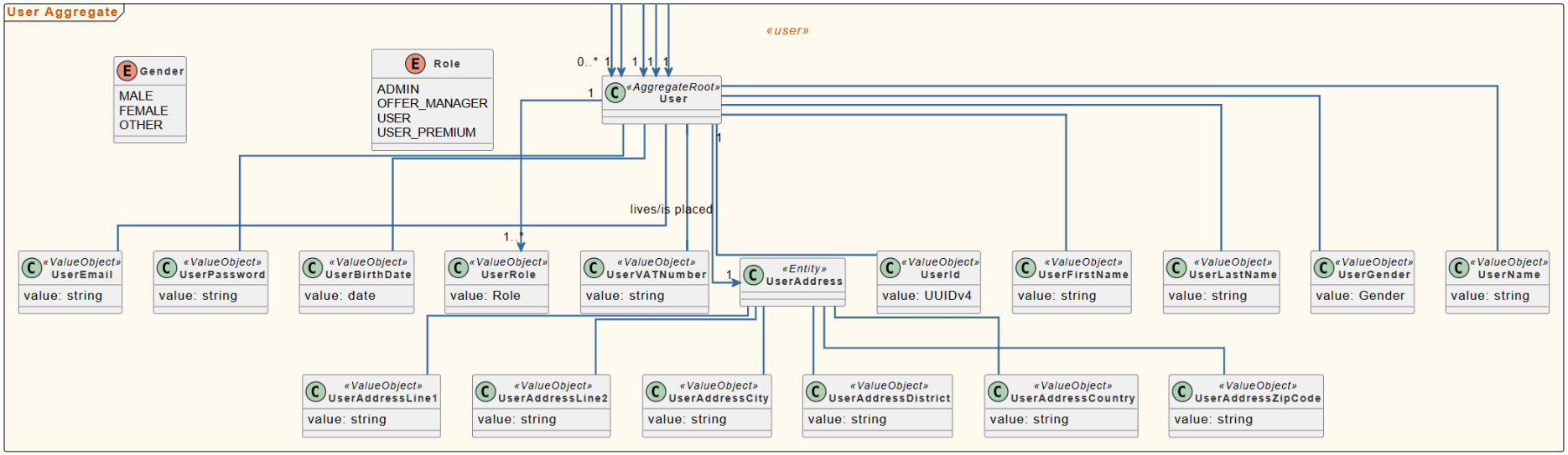


Figure 52 - Domain Model - User Aggregate

The Points Earn Transaction Aggregate is the simplest aggregate on the application since the project owners have not yet exploited this gamification system, and it is represented in Figure 53.

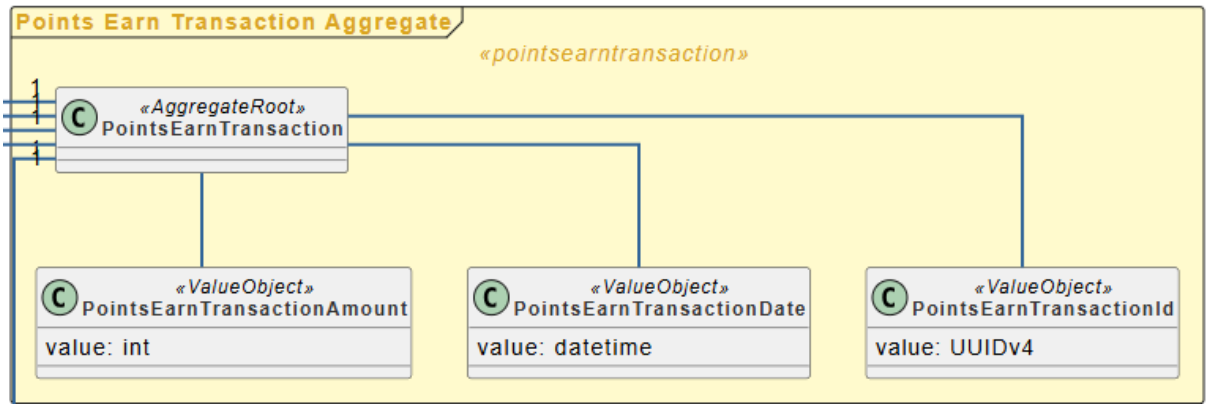


Figure 53 - Domain Model - Points Earn Transaction Aggregate

The Wifi Spot Aggregate is the most complex in the application. It is the meaning of the project's existence, being very exploited. It can be seen in Figure 54 and Figure 55.

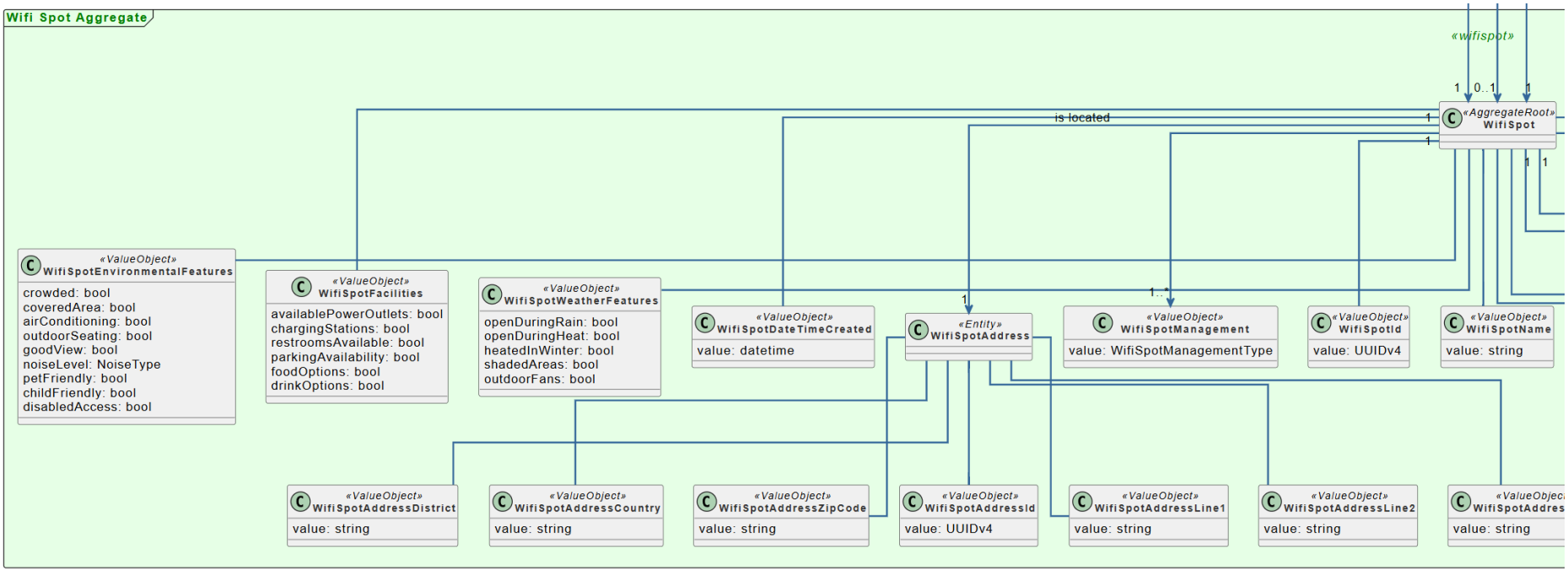


Figure 54 - Domain Model - Wifi Spot Aggregate (1)

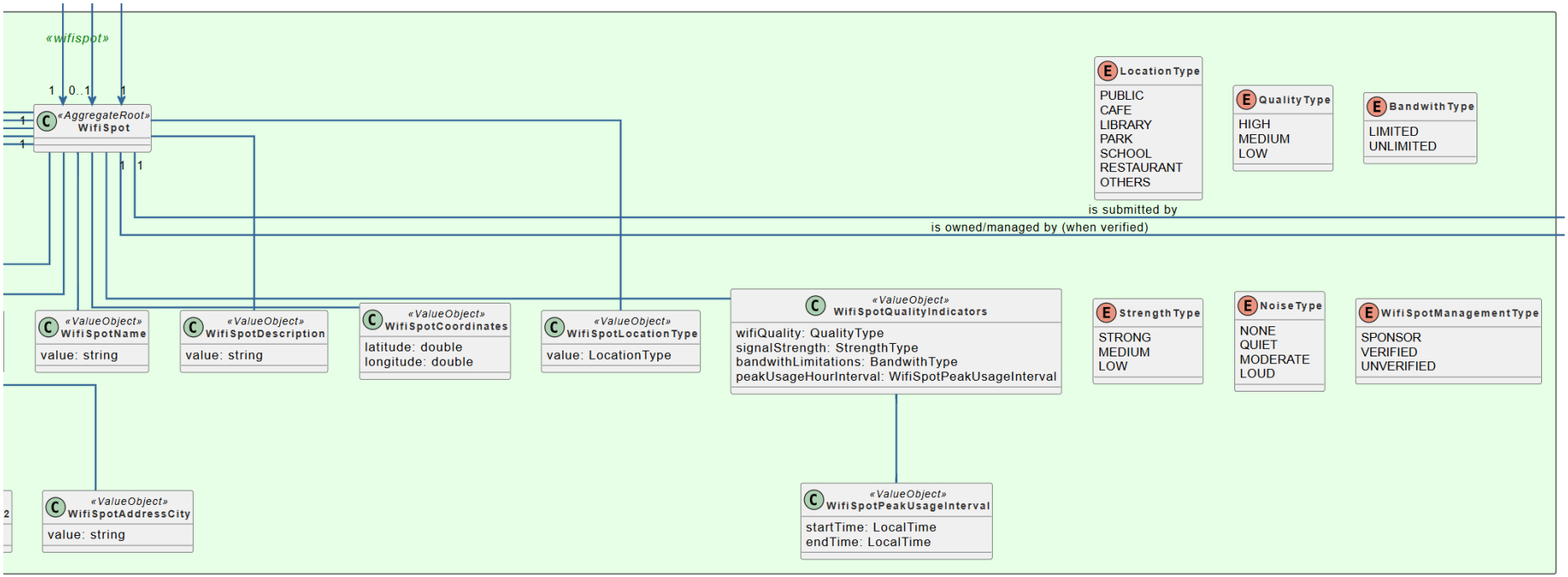


Figure 55 - Domain Model - Wifi Spot Aggregate (2)

Appendix D

This appendix contains information about the process and implementations to install Grafana k6.

One simple command was executed in the Command Line: *winget install k6 --source winget* [86]. After the installation, five JavaScript files were developed that import the k6 libraries and do the performance analysis, one for each scenario [87]. One example can be seen in Code Snippet 14.

```
import http from 'k6/http'
import { check, sleep } from 'k6'
import encoding from 'k6/encoding'

//k6 run .\create_review_scripts.js

var database = "mongodb"

export let options = {
    vus: 1,
    iterations: '5',
}

export default function () {
    const username = 'admin'
    const password = 'admin'
    const credentials = `${username}:${password}`
    const encoded = encoding.b64encode(credentials)

    const res =
http.post('http://localhost:8080/api/review/'+database+'/file-
create', null, {
        headers: {
            'Authorization': `Basic ${encoded}`
        }
    })

    check(res, {
        'status is 200': (r) => r.status === 200
    })

    sleep(0.3)
}
```

Code Snippet 14 - Create a review k6 JavaScript script

The *options* variable defines that only one Virtual User (one thread) is going to be executed, and the iterations specify the number of times the function will be executed on each Virtual User. The iterations are set to five because it is essential to have an average value of results, and not a single result that could be affected by external processes in the system.

To execute the script, it is necessary to run it in the command line: *k6 run [script].js*. The result is shown in Figure 56 where it is possible to see the "*avg*" time, followed by the "*min*", "*max*", among others.

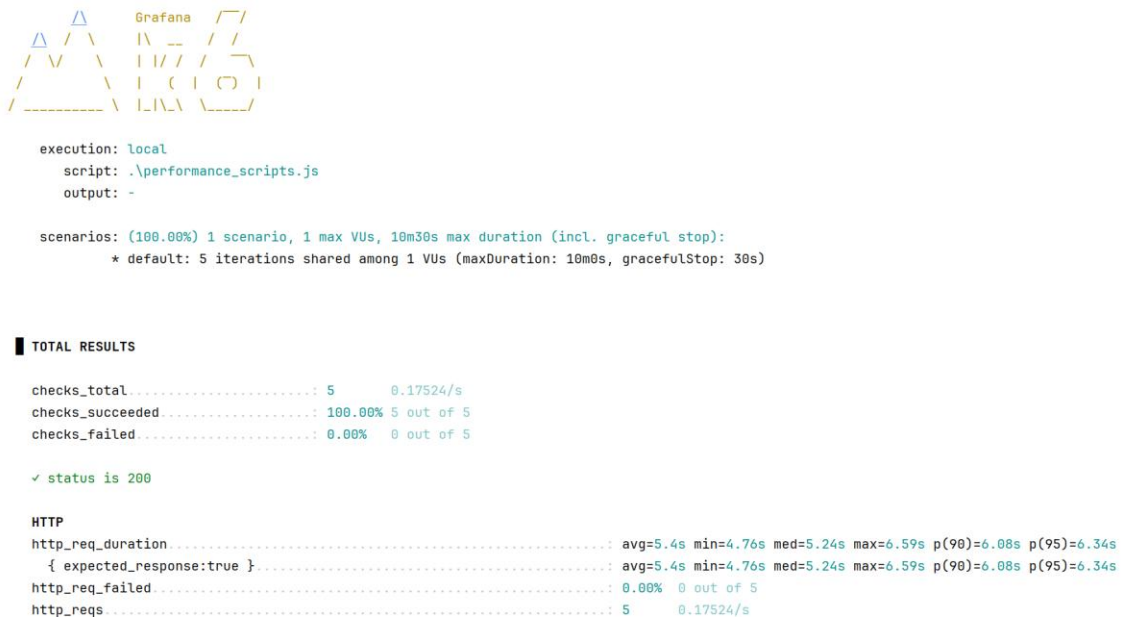


Figure 56 - Grafana k6 result analysis example

Appendix E

Some problems were encountered when implementing Kepler in the application.

Initially, it was deployed to a Docker Kubernetes server. However, Docker Kubernetes has a limitation for Kepler measurements; some files used to measure CPU usage are unavailable in windows operatives, causing invalid Kepler validations.

Therefore, a Virtual Machine (VM) was configured in Oracle VirtualBox. An Ubuntu machine on which the Ubuntu 24.04.2 LTS ISO was installed. Then, all the configurations and installations were performed:

- Java.
- Docker and Docker-compose.
- Python and Faker.
- Git.
- Maven.
- K3S and Helm for Kubernetes.
- Kepler.

Having all this installed, the Java application was then deployed to the Kubernetes server in k3s, using a Dockerfile present in Code Snippet 15.

```
FROM openjdk:17-jdk

COPY target/netquest-api-0.0.1-SNAPSHOT.jar /app.jar

ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Code Snippet 15 - Deployment of the Java application – Dockerfile

After some configurations for Kubernetes to know which application will be running, using *.yaml* files, all the previous developments were made again on this Virtual Machine, for example, the deployment of the databases into the Kubernetes server.

Finally, five scripts were done, one for each scenario, as was done for Grafana k6. These scripts call the k6 scripts and compare the Joules consumption before and after the execution. This way, it is possible to have performance and energy results in one single script. One example can be seen in Code Snippet 16, with an example of an output in Figure 57.

```
#!/bin/bash

echo "Capturing Kepler metrics before test"
curl -s http://localhost:9102/metrics | grep netquest-api >
before.txt

echo "Running k6 load test..."
k6 run wifi_spot_name_scripts.js

sleep 2

echo "Capturing Kepler metrics after test"
curl -s http://localhost:9102/metrics | grep netquest-api >
after.txt

echo "Comparing energy and resource usage:"
diff before.txt after.txt
```

Code Snippet 16 - Script for energy consumption

Figure 57 - Example of energy consumption results