JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 33 (2024) 298–305
Received: 21 June 2024
Accepted: 10 July 2024

Performance analysis of working with relational and non-relational databases in Java applications

Analiza wydajności pracy z relacyjnymi i nierelacyjnymi bazami danych w aplikacjach Java

Krzysztof Caban*, Paweł Czuchryta, Beata Pańczyk

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The article presents a performance analysis of connections to both relational and non-relational databases, critical components of the functionality of modern web applications. The study is concerned with evaluating the advantages of manually integrated database drivers compared to the comprehensive Spring Data module. In addition, the impact of the Spring Framework on the performance of drivers responsible for database connections was investigated. Based on the results obtained, there are performance benefits for CRUD operations when adding drivers manually and using the Spring Framework for JDBC and MongoDB drivers.

Keywords: Spring Framework; Java; performance; database drivers

Streszczenie
Artykuł przedstawia analizę wydajności połączeń zarówno z relacyjnymi, jak i nierelacyjnymi bazami danych, krytycznymi komponentami funkcjonalności współczesnych aplikacji internetowych. Badanie dotyczy oceny zalet ręcznie zintegrowanych sterowników baz danych w porównaniu do kompleksowego modułu Spring Data. Ponadto zbadano wpływ Spring Framework na wydajność sterowników odpowiedzialnych za połączenia z bazami danych. Na podstawie uzyskanych wyników stwierdzono, że istnieją korzyści w zakresie wydajności dla operacji CRUD w przypadku ręcznego dodawania sterowników i korzystania ze Spring Framework dla sterowników JDBC i MongoDB.

Słowa kluczowe: Spring Framework; Java; wydajność; sterowniki bazodanowe

*Corresponding author
Email address: krzysztof.caban@pollub.edu.pl (K. Caban)

1. Introduction

Nowadays, databases are crucial for the operation of computer programs, particularly web-based applications that now dominate the world. Most of these applications store data; therefore, databases are critical for proper operation. Developments in technology and development tools lead to an increase in user demands on applications for faster data processing, which makes the performance of the connection between the application and the database important. This interfacing is made possible by the use of dedicated drivers, which act as an interface between the application and database.

Currently, one of the critical criteria used to classify databases is by their structure either relational or non-relational. The decision to use either a relational or a non-relational database is naturally informed by the nature of the information being held. Relational databases are used where data to be stored has well-defined and structural relationships plus the requirement to carry out transactions to maintain consistency. An excellent example of such a database, based on relational storage, is a well-known MySQL [1] system. Non-relational databases are preferred in operation cases with massive datasets, the variability of patterns, priority of scalability, flexibility, and speed of access to information. MongoDB [2] is one of the very well-known databases with non-relational storage mechanisms [3].

In the case of database connection drivers, they can be used either as discrete libraries or more extensive modules consisting of a wide range of other different drivers in implementing one. For instance, in the writing of a Java [4] application, they can be included in the project using a tool, such as the Maven. Developed applications based on the Spring [5] development framework can include the Spring Data [6] module as a set of tools and libraries for connecting to relational and non-relational databases.

The paper discusses the problem of optimal usage in terms of connection to the relational and non-relational databases. The study will examine the advantages of manually integrated database drivers against the benefits of selecting the more robust Spring Data module. These studies are related to whether use of the Spring framework positively affects drivers responsible for connecting to databases.

2. Related work

This chapter provides a review of the available literature on the comparative analysis of programming frameworks used in Java and Spring applications to communicate with relational and non-relational databases. Given the limited number of publications on this topic, it was chosen as the subject of this paper.

In the article "Spring Framework Reliability Investigation Against Database Bridging Layer Using Java Platform" a comparative analysis of development frameworks used to communicate with relational databases was conducted [7]. The study tested the performance of Hibernate [8], Java Database Connection (JDBC) [9] and MyBatis Framework using a customized web service written in Spring. The study tested only the operation of obtaining data from the application using a prepared test script. Based on the results, it was possible to determine that the MyBatis and Hibernate frameworks achieved comparable results, while JDBC proved to be the least efficient solution.

In the publication "Performance Evaluation of Transparent Persistence Layer in Java Applications" authors compared performance between Apache Object Relational Bridge (OJB), Hibernate and Hibernate combined with Spring [10]. The study tested CRUD (create, read, update, delete) operations for a single database and a distributed solution. Tests were performed for a range of objects from 0 to 1000 with increment by 50, and the completion time of the operations was compared. The tests showed that the combination of Hibernate and Spring provided the best stability and performance of the operations performed at the expense of higher memory usage caused by the use of the two frameworks.

The publication "Multi-Platform Performance Analysis for CRUD Operations in Relational Databases from Java Programs using Spring Data JPA" focuses on evaluating the performance of several leading relational database management systems in the context of interaction with Java applications using the JPA development framework [11]. The research focused on comparing the performance of CRUD operations. Tests were performed for different record counts 1000, 10000, 50000, in order to identify the optimal database depending on the usage scenario. Analysis of the results showed that the MySQL database proved to be the most versatile. SQLServer, on the other hand, achieved the highest performance in the context of data read operations. Interesting conclusions emerged in the context of the Oracle database, which obtained results below the average presented by other solutions. This analysis provides information to make a decision on the database used based on the type and number of operations.

The articles focused either on comparing performance between different databases based on one of the available communication solutions or pay attention to a single communication method and present its capabilities. Therefore, the present study focuses on comparing different communication methods with relational and non-relational databases.

3. Material and methods

For this study, a comprehensive analysis of the performance of various database connection solutions was carried out. To get the full picture, several versions of the application were prepared using different configurations. Applications were prepared in the default driver configuration without any changes that might affect the optimisation.

When working with relational databases, the following configurations were used:

- JDBC in a Java application,
- JDBC in a Java + Spring application,
- Hibernate in a Java application,
- Hibernate in a Java + Spring application,
- Application with Spring Data JDBC module,
- Application with Spring Data JPA module.

In the context of performance testing of non-relational database connection solutions, the following approaches were used:

- MongoDB [12] driver in a Java application,
- MongoDB driver in a Java + Spring application,
- Application with the Spring Data MongoDB module.

To test the application, relational and non-relational database schemas were developed to store information about the furniture wholesaler's product range. As depicted in Figure 1, the relational database schema is structured into four tables. The primary table, 'furniture', holds detailed information about each piece of furniture. Complementary tables include the 'manufacturer' table, which catalogues information about the manufacturers, the 'type' table, which classifies the types of furniture, and the 'material' table, which describes the materials used in the furniture. For comprehensive testing, the database was populated with 10,000 records.

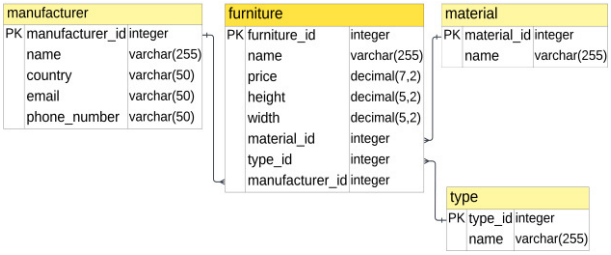

Figure 1: Database schema.

3.1. Testing scenarios

In order to benchmark performance, several scenarios were prepared which were executed for each of the prepared application configurations. The tests checked the execution time of HTTP requests and the use of CPU and RAM. The test scenarios were prepared in JMeter.

Test scenarios I-V for HTTP requests were prepared and run for different numbers of query repetitions of 1, 10, 100, 1 000 and 10 000 by one user:

- Scenario I – GET method – the user downloads all available records in the database.
- Scenario II – GET method – the user retrieves a single record.
- Scenario III – POST method – the user creates a new record.
- Scenario IV – PUT method – the user edits a single record in the database based on id.
- Scenario V – DELETE method – the user deletes a single record from the database.

In order to measure the consumption of resources, additional test scenarios were prepared. Each of them was executed for 2 minutes continuously and with different numbers of concurrent users 1, 10 and 100, respectively – at a constant throughput of 150 requests per second.

- Scenario VI – GET method – downloading a single record from the database.
- Scenario VII – POST method – adding a new record to the database.

3.2. Test environment

Docker [13] containers were used to make the application operation independent of the platform on which the tests were executed. The structure included 2 Docker containers:

- Java application container,
- Database container.

The hardware specifications used in this research can be seen in Table 1.

Table 1: Test platform specification

Component	Description
CPU	Intel Core i5-13400F
RAM	32GB DDR4
Operating system	Windows 11 Home
Java	21
Spring Boot	3.2.4
JDBC	8.3.0
Hibernate	6.4.4
MongoDB driver	4.11.2
MongoDB	7.0.7
MySQL	8.4.0

4. Results

This chapter presents the results obtained from the research conducted on the performance analysis of database connections, both relational and non-relational. These results are presented in the form of graphs, showing the average measurement values for the different test scenarios. To facilitate the analysis, the data has been divided into subsections, each focusing on a specific database connection driver. This division allows the performance of different solutions to be compared, i.e. manually attaching drivers versus off-the-shelf solutions, and the impact of using the Spring Framework on the performance of the drivers tested. In average test time graphs, different colours represent the number of queries performed during a single test. In resource usage graphs, colours represent the number of concurrent users during the test.

4.1. JDBC driver

Figures 2-6 show the average times obtained in scenarios I-V for applications using the JDBC driver. The worst times can be observed in the Spring Data JDBC module application, while the best times were observed in the Spring application with a manually supplied JDBC driver.

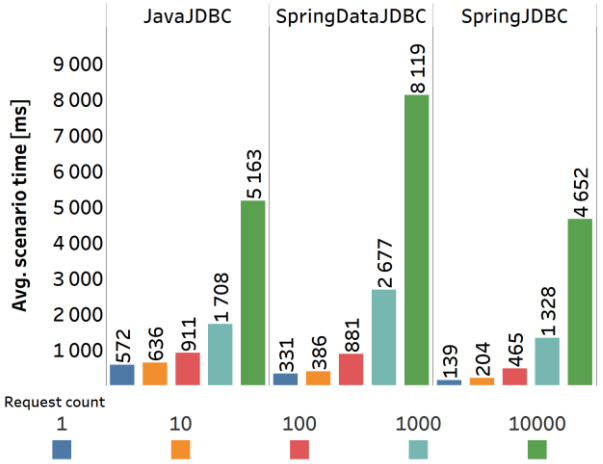

Figure 2: Average execution time in Scenario I (GET ALL).

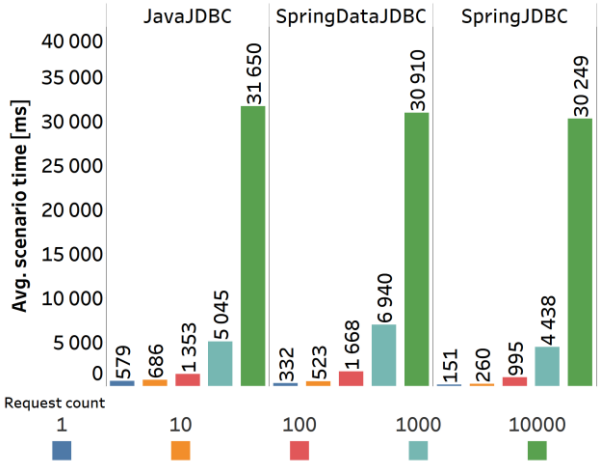

Figure 3: Average execution time in Scenario II (GET ONE).

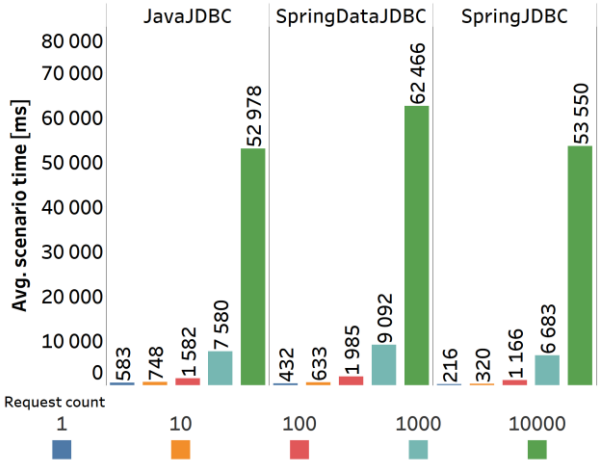

Figure 4: Average execution time in Scenario III (POST).

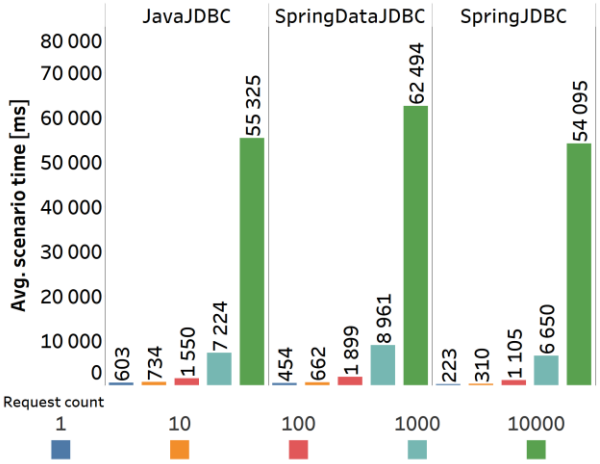

Figure 5: Average execution time in Scenario IV (PUT).

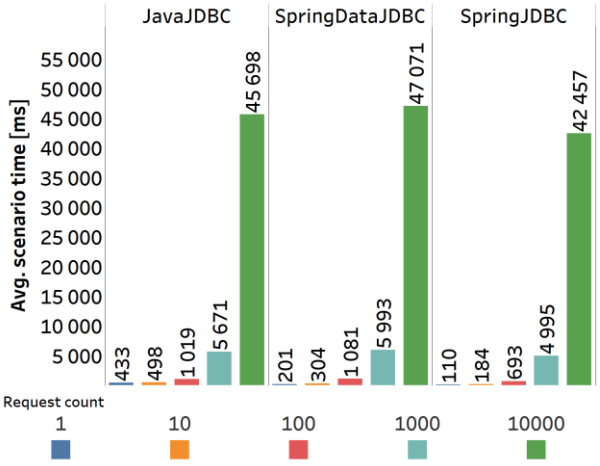

Figure 6: Average execution time in Scenario V (DELETE).

Figures 7-10 show the average resource consumption (RAM and CPU) in scenarios VI-VII for applications using the JDBC driver. It can be observed that the overall resource consumption of applications with manually added drivers is lower than the off-the-shelf Spring Data JDBC solution. The lowest average resource consumption can be distinguished here by the Spring + manually added JDBC application.

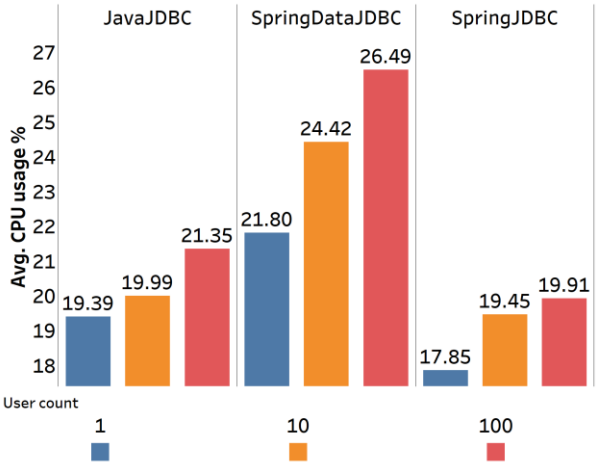

Figure 7: Average CPU usage in Scenario VI (GET).

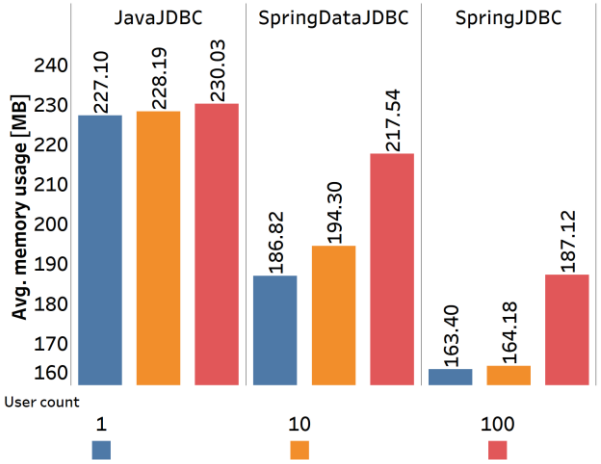

Figure 8: Average RAM usage in Scenario VI (GET).

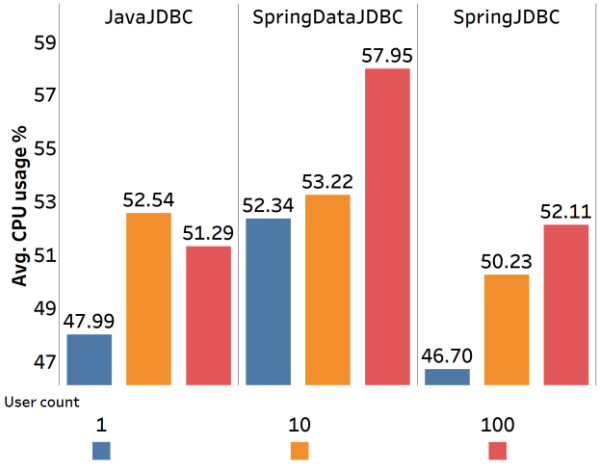

Figure 9: Average CPU usage in Scenario VII (POST).

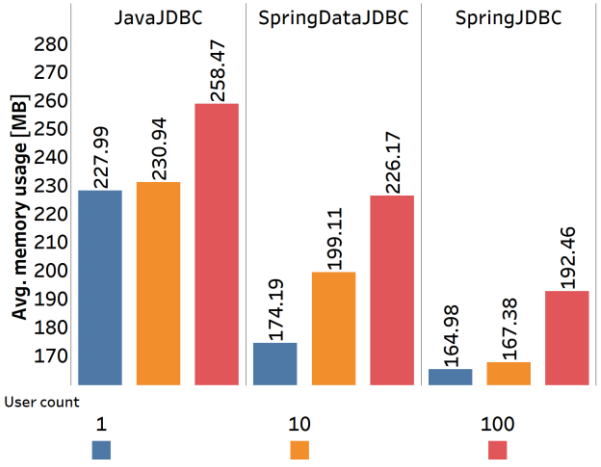

Figure 10: Average CPU usage in Scenario VII (POST).

4.2. Hibernate driver

Figures 11-15 show the average times obtained in scenarios I-V for applications using the Hibernate driver. In the case of the Hibernate driver, the best times could be observed for the application with the off-the-shelf Spring Data JPA module. By far the worst in these tested scenarios was the Spring application with a manually attached Hibernate driver.

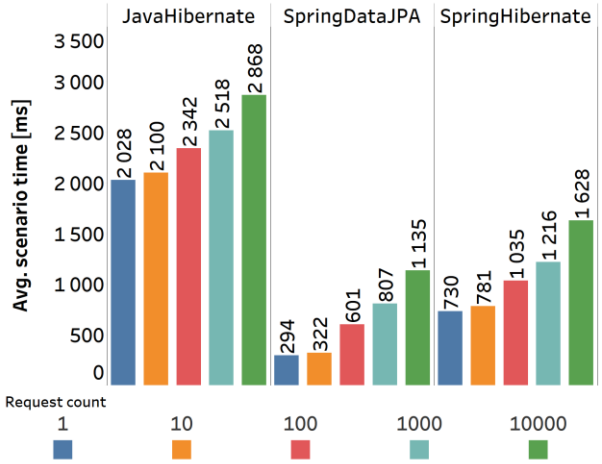

Figure 11: Average execution time in Scenario I (GET ALL).

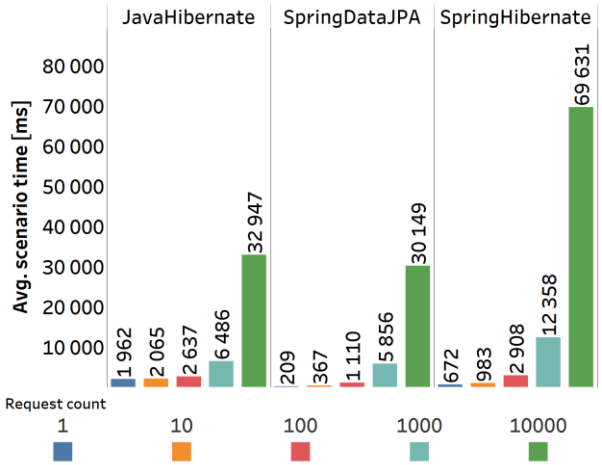

Figure 12: Average execution time in Scenario II (GET ONE).

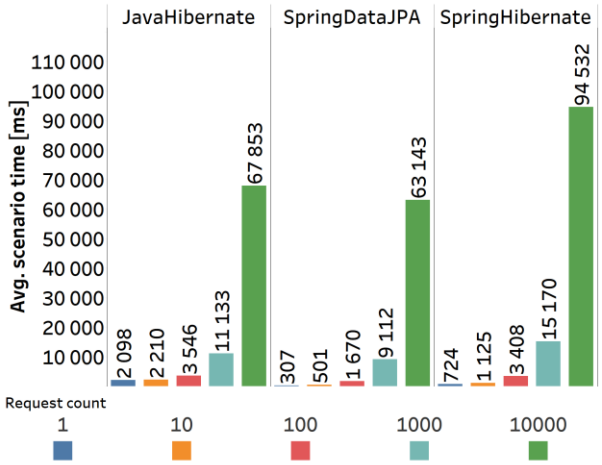

Figure 13: Average execution time in Scenario III (POST).

Figures 16-19 show the average resource consumption (RAM and CPU) in scenarios VI-VII for applications using the Hibernate driver. Again, the application with the ready-made Spring Data JPA module performed best and showed the lowest resource consumption. Also, as in scenarios I-V, the Spring application with manually added Hibernate proved to be the worst one and showed the highest average resource consumption, especially in the test case with 100 users.

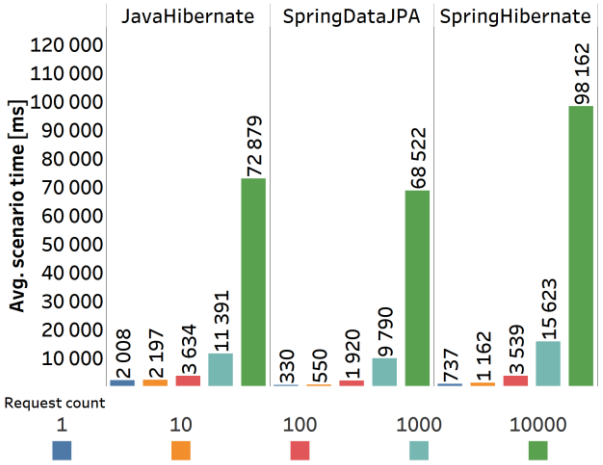

Figure 14: Average execution time in Scenario IV (PUT).

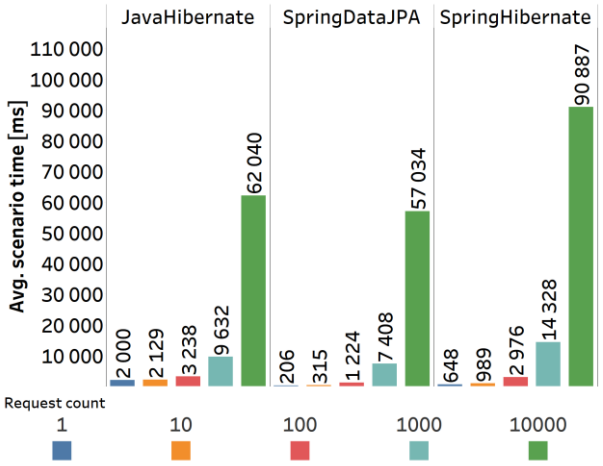

Figure 15: Average execution time in Scenario V (DELETE).

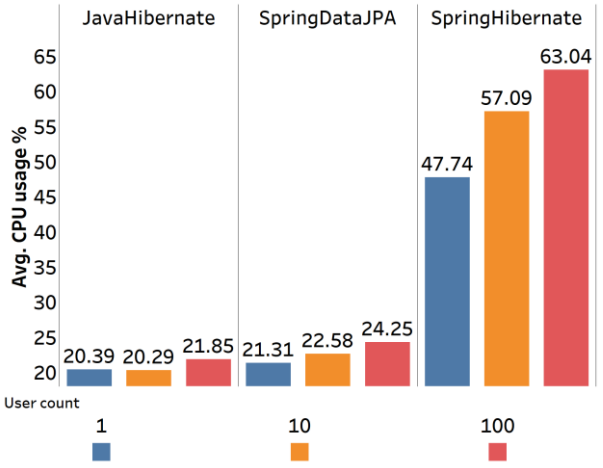

Figure 16: Average CPU usage in Scenario VI (GET).

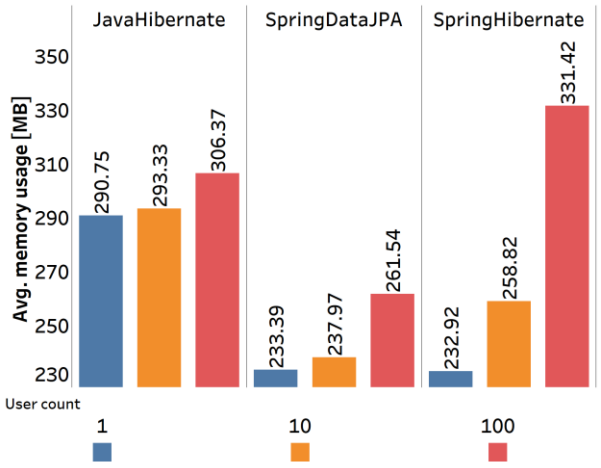

Figure 17: Average RAM usage in Scenario VI (GET).

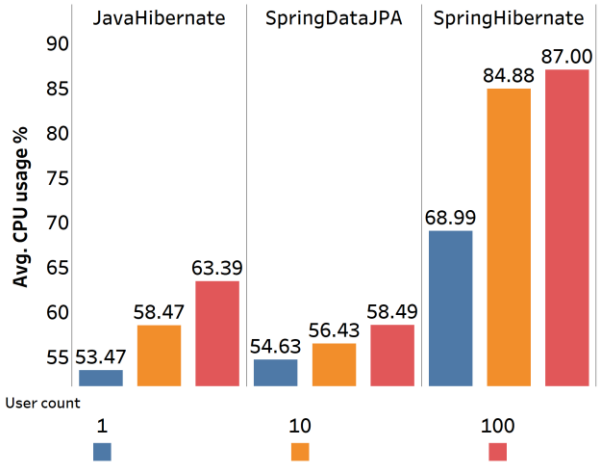

Figure 18: Average CPU usage in Scenario VII (POST).

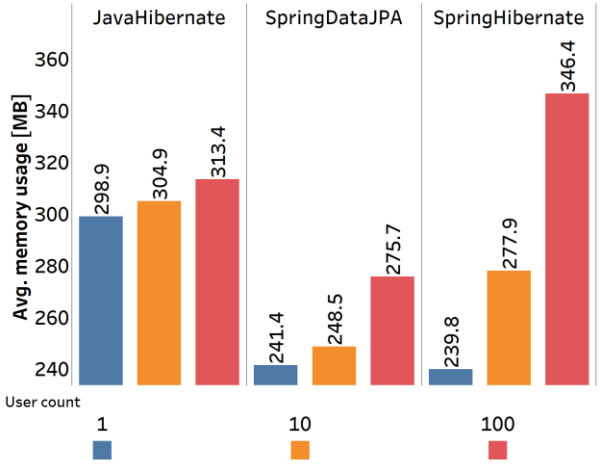

Figure 19: Average RAM usage in Scenario VII (POST).

4.3. MongoDB driver

Figures 20-24 show the average times obtained in scenarios I-V for applications using the MongoDB driver. Very similar to the situation with the JDBC driver, the applications with the manually added MongoDB driver showed the lowest average times in the tests performed. The worst application was the one with the ready-made Spring Data MongoDB solution, and the best was the Spring application with the manually attached driver.

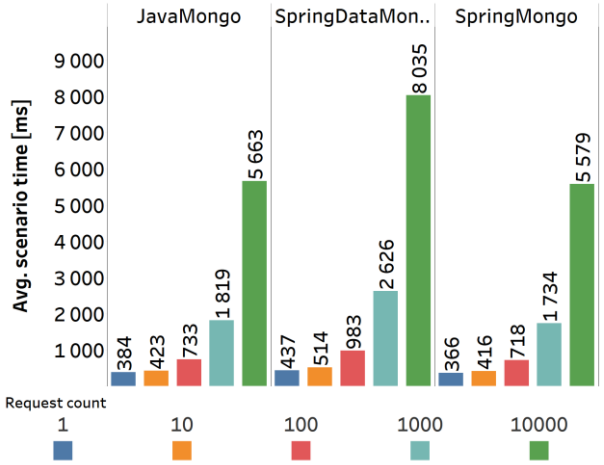

Figure 20: Average execution time in Scenario I (GET ALL).

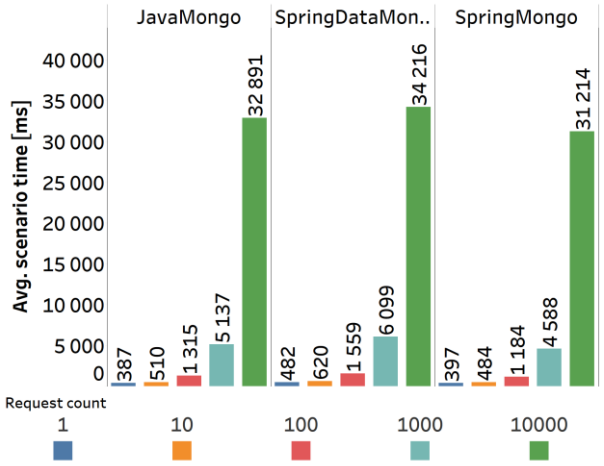

Figure 21: Average execution time in Scenario II (GET ONE).

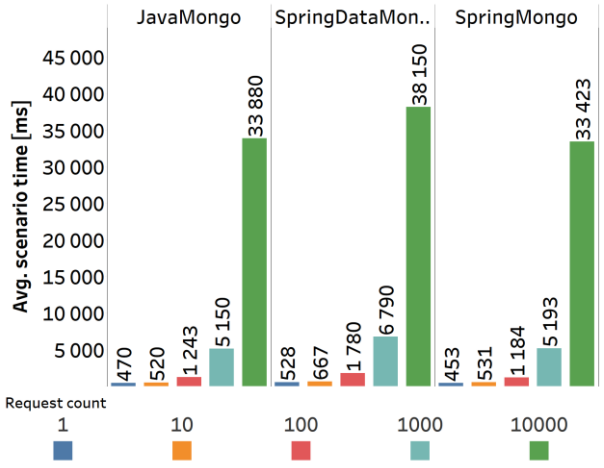

Figure 22: Average execution time in Scenario III (POST).

Figures 25-28 show the average resource consumption (RAM and CPU) in scenarios VI-VII for applications

using the MongoDB driver. In the performance tests, the Spring application with the manually added MongoDB driver also won, showing the lowest average consumption of RAM and CPU resources.

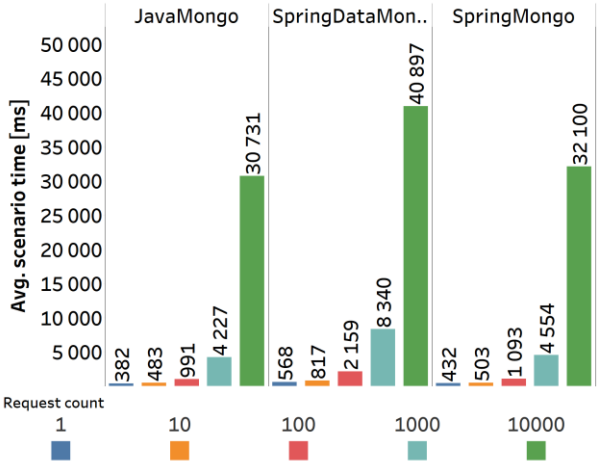

Figure 23: Average execution time in Scenario IV (PUT).

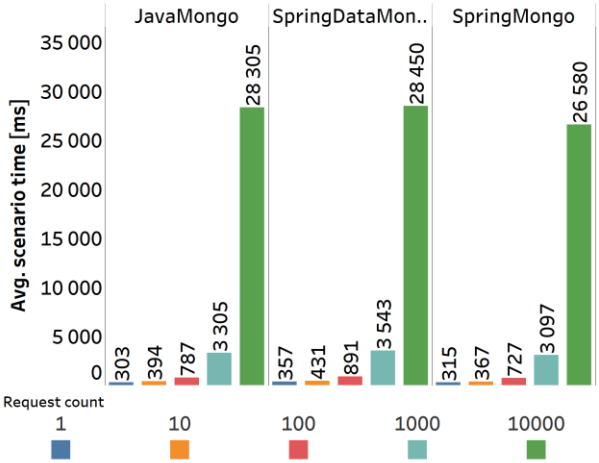

Figure 24: Average execution time in Scenario V (DELETE).

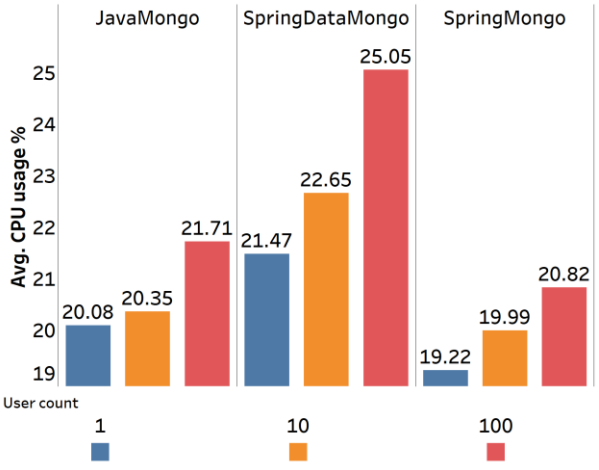

Figure 25: Average CPU usage in Scenario VI (GET).

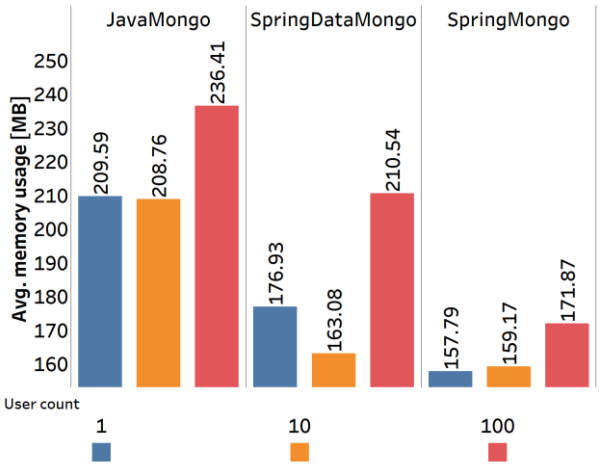

Figure 26: Average RAM usage in Scenario VI (GET).

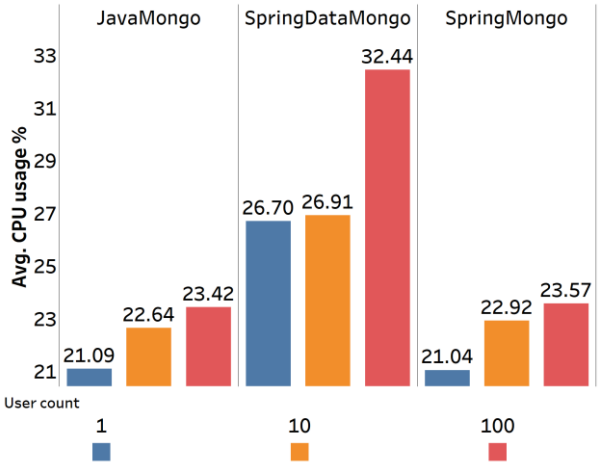

Figure 27: Average CPU usage in Scenario VII (POST).

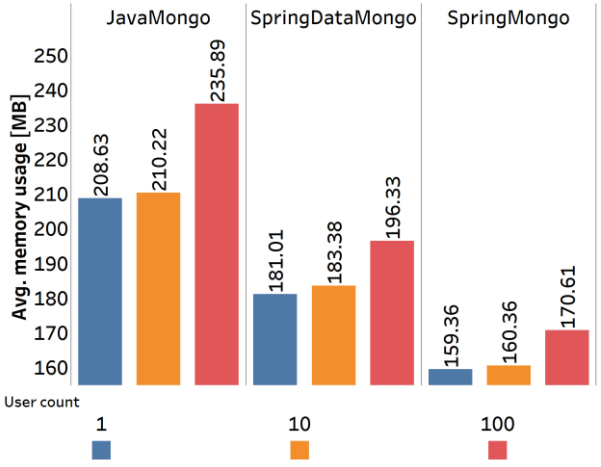

Figure 28: Average RAM usage in Scenario VII (POST).

5. Conclusion

From the results, it can be concluded that, for two out of three drivers, manual integration provided better performance than the out-of-the-box solution provided by the Spring Data module. Performance improvement was also observed in drivers integrated into the context of an

application based on the Spring Framework, which implies a positive impact on efficiency of solution.

The results affirmed that the applications which used the Spring framework and manually attached driver, performed better in terms of response time and resource consumption compared to that of applications written in Java and Spring with use of Spring Data module. (14 of the tested 21 variants). Exception was the Spring Data JPA module, which ran faster than its counterparts.

The other important finding was that the worst performance results among those obtained were created by combining a Hibernate manually added driver with an application based on the Spring Framework. This is likely because all the solutions were tested in their default configuration.

It is also important to remember that such drivers can be further adjusted to improve the obtained performance of applications. Further research could attempt to find the best possible configuration for tested drivers. Once completed, further tests might be performed in order to select the best configuration in comparison to the Spring Data module. This process would provide researchers with valuable data in assessment of the value of a custom-made data layer in comparison to well-known and established parts of Spring Framework.

References

[1] MySQL, https://www.mysql.com/, [08.11.2023].

[2] MongoDB: The Developer Data Platform, https://www.mongodb.com/, [08.11.2023].

[3] Best NoSQL Databases Software in 2023, https://6sense.com/tech/nosql-databases, [08.11.2023].

[4] Java| Oracle, https://www.java.com/pl/, [08.11.2023].

[5] Introduction to Spring Framework, https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html, [08.11.2023].

[6] Spring Data, https://spring.io/projects/spring-data, [07.12.2023].

[7] A. Ginanjar, M. Hendayun, Spring Framework Reliability Investigation Against Database Bridging Layer Using Java Platform, Procedia Computer Science 161 (2019) 1036-1045, https://doi.org/10.1016/j.procs.2019.11.214.

[8] Your relational data. Objectively. - Hibernate ORM, https://hibernate.org/orm/, [08.11.2023].

[9] Java JDBC API, https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, [08.11.2023].

[10] Z. Zhiyu, C. Zhiang, Performance Evaluation of Transparent Persistence Layer in Java Applications, Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (2010) 21-26, https://doi.org/10.1109/CyberC.2010.15.

[11] A. M. Bonteanu, C. Tudose, A. M. Anghel, Multi-Platform Performance Analysis for CRUD Operations in Relational Databases from Java Programs using Spring Data JPA, Proceedings of the 13th International Symposium on Advanced Topics in Electrical Engineering (ATEE) (2023) 1-6, https://doi.org/10.1109/ATEE58038.2023.10108212.

[12] Start Developing with MongoDB - MongoDB Drivers, https://www.mongodb.com/docs/drivers/, [17.01.2024].

[13] Docker Documentation: Docker overview, https://docs.docker.com/get-started/overview/, [08.11.2023].