JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 35 (2025) 129–136
Received: 09 January 2025
Accepted: 11 February 2025

Comparison of data processing efficiency in Java and Scala

Porównanie efektywności przetwarzania danych w językach Java i Scala

Bartosz Markiewicz*, Krzysztof Matyjaszczyk, Marek Miłosz

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

Abstract

This article compares the efficiency of Java and Scala languages in data processing. Two web applications were created in these languages using the popular GraphQL interface and the H2 database. The research was conducted using test scenarios that included several data processing algorithms, varying numbers of users making requests to the application and different database sizes. Performance evaluation criteria such as data processing time, memory usage, and processor load were applied. The JMeter tool was used for the tests. The results show that the application written in Scala demonstrates higher efficiency.

Keywords: Java; Scala; data processing; efficiency

Streszczenie

W niniejszym artykule dokonano porównania efektywności języków Java i Scala przy przetwarzaniu danych. Utworzono dwie aplikacje internetowe w tych językach z wykorzystaniem popularnego interfejsu GraphQL oraz bazy danych H2. Badania wykonano dla scenariuszy testowych uwzględniających kilka algorytmów przetwarzania danych, różną liczbę użytkowników wykonujących żądania do aplikacji oraz wielkość bazy danych. Wykorzystano takie kryteria oceny efektywności jak czas przetworzenia danych, wykorzystanie pamięci operacyjnej oraz obciążenie procesora. Do przeprowadzenia badań wykorzystano narzędzie JMeter. Wyniki wykazują, że aplikacja napisana w języku Scala charakteryzuje się większą efektywnością.

Słowa kluczowe: Java; Scala; przetwarzanie danych; efektywność

*Corresponding author

Email address: **bartosz.markiewicz@pollub.edu.pl** (B. Markiewicz)

1. Introduction

Java [1] has remained one of the most frequently used languages in business software development for many years [2]. Developers appreciate its rich ecosystem of libraries and tools, as well as its stability, long-term support, and simplicity of purely object-oriented code [3]. A key element of its popularity and versatility is the Java Virtual Machine (JVM) [4], which serves as a universal platform for running applications across different environments. In recent years, alternatives to Java based on the JVM, such as Kotlin and Scala, have gained popularity. Scala [5], combining object-oriented and functional programming features, is becoming increasingly attractive in professional environments [6]. Thanks to its advantages, it is now widely used in projects ranging from big data systems to data streaming platforms and backend applications, making it an essential part of the modern technological landscape.

Developers value the JVM platform for its stability, reliability, and ability to run applications in diverse environments [3]. Choosing the right programming language for developing a business system is one of the key challenges software architects face [7]. This decision depends on various factors, including the language's efficiency, compatibility with the existing ecosystem, and the availability of qualified developers. Community support and the long-term development perspective of the selected language are also crucial aspects. Therefore, selecting the right language requires a careful analysis of both the technical and business needs of the project.

This article compares the efficiency of two JVM platform languages, Java and Scala. The comparison focuses on the most common type of business software — web applications [7]. Two web applications will be developed in these languages [8]. These applications will communicate via the Hypertext Transfer Protocol (HTTP), and the processed data will be stored in a file-based database. The conducted research will consider various system load scenarios. Three algorithms with different data processing approaches will be used. The applications will be powered by data volumes of varying sizes, from small to those containing hundreds of thousands of records. Additionally, an analysis of the impact of the number of concurrent users on application performance will be conducted.

Before starting the research, the following thesis has been proposed: **Scala is a more efficient solution for data processing in web applications**. To confirm this thesis, the following hypotheses have been formulated:

H1. Scala processes both small and large data volumes faster.
H2. Scala processes data faster using multithreading.
H3. Scala uses less memory during data processing.
H4. Scala causes less processor load during data processing.

Conducting the research and analyzing the results will lead to the verification of the proposed thesis and hypotheses.

2. Literature review

Performance studies of web applications are the subject of numerous scientific publications. These studies often compare various programming languages, frameworks, and libraries. Publications discussing specific techniques for testing application performance and tools for measurements are also common.

In the article "A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core" [9], the authors compared the efficiency of two web applications. To ensure reliable research, a key aspect was developing applications with identical functionalities, despite being implemented in two different languages. The study used several metrics to compare performance, such as response time or memory and processor usage.

The author of the publication "Performance analysis of languages working on Java Virtual Machine based on Java, Scala and Kotlin" [10] conducted a comparative analysis of JVM-based languages. For this purpose, applications were developed in Java, Scala, and Kotlin. These applications calculated Fibonacci sequence terms and evaluated regular expressions. The results showed no significant performance differences between the languages; however, it was noted that Scala often achieved the weakest results.

In the article "Comparative analysis of web application performance testing tools" [11], the authors reviewed and compared popular tools for testing the performance of web applications: Apache JMeter, LoadNinja, and Gatling. Apache JMeter stood out as a noteworthy tool due to its graphical user interface, free access, and high popularity among users.

The literature review enabled the authors to acquire knowledge about web application performance testing. A critical consideration is ensuring that applications developed in different languages or frameworks provide identical functionalities and that their algorithms are as structurally similar as possible. Key criteria for evaluating application performance are execution time and hardware resource usage. It is worth noting that authors of publications comparing the efficiency of JVM platform languages often report no significant differences between the languages. Based on the analysis of publications about available tools, the authors of this article decided to use Apache JMeter to conduct performance tests of the implemented applications.

3. Research methods

The goal of the research is to determine which programming language is more efficient in the context of data processing. For this, two web applications written in Java and Scala, offering identical functionalities, were tested. The performance analysis employed four criteria. The technical specification of each criterion is as follows:

- Processing time at the controller level – the request processing time by the GraphQL controller, including fetching data from the database and then processing it, measured in milliseconds [ms].
- Processing time at the algorithm level – the execution time of the data processing algorithm with prefetched data as an input to the algorithm, measured in milliseconds [ms].
- Memory usage – the maximum JVM heap size during requests execution, measured in mega bytes [MB].
- Processor utilization – the maximum computational power usage of the processor by the JVM during request execution, measured as a percentage of utilization [%].

To conduct the study, two applications were developed with functionalities for data processing within a simplified model of an online store's order management. The Java application was built using the Spring framework [12], while the Scala application utilized the Akka framework [13]. The ERD (Entity relationship diagram) diagram of the data model used in the applications is shown in Figure 1.

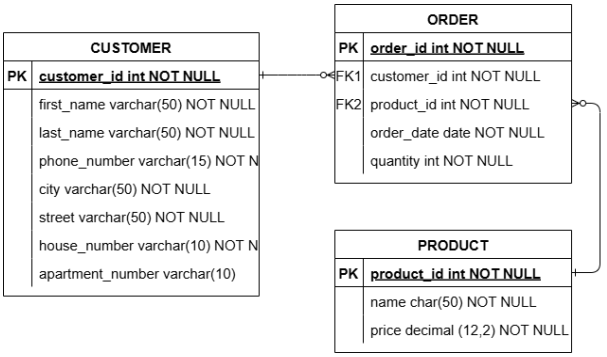

Figure 1: ERD diagram of data model used in applications.

The implemented data processing algorithms are outlined in Table 1.

Table 1: Methods implementing algorithms used during research

A1	Find the customer with most orders
A2	Find the most sold product
A3	Calculate the total value of all orders

Code fragments [14] from the Java and Scala applications implementing the A3 algorithm are presented in Listings 1 and 2.

Listing 1: Algorithm A3 implemented in Java

```
public BigDecimal calculateTotalOrdersValue() {
    List<Order> orders = getAllOrders();
    long start = System.nanoTime();

    BigDecimal totalOrdersValue = orders.stream()
            .map(order -> order.product().price()
                    .multiply(new BigDecimal(
                            Integer.toString(order.quantity()))))
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    long end = System.nanoTime();
    log.info("OrderDao.calculateTotalOrdersValue(): %dns"
            .formatted( ...args: end - start));

    return totalOrdersValue;
}
```

Listing 2: Algorithm A3 implemented in Scala

```
def calculateTotalOrdersValue: BigDecimal = {
  val orders = getAllOrders
  val start = System.nanoTime()

  val totalOrdersValue = orders
    .map(order => order.product.price
      * BigDecimal.valueOf(order.quantity))
    .reduceOption(_ + _)
    .getOrElse(BigDecimal(0))

  val end = System.nanoTime()
  logger.info(s"OrderDao.calculateTotalOrdersValue(): " +
    s"${end - start}ns")

  totalOrdersValue
}
```

For communication with the applications, the modern and widely used query language GraphQL [15] was employed. The data processed by the applications was stored in a local file-based H2 database. The data volumes used to populate the database were randomly generated using Python scripts with the Faker library. To measure hardware resource usage, the Prometheus tool was used, allowing the visualization of resource consumption trends over time. Performance tests were conducted using Apache JMeter software. The test environment was set up on a VirtualBox virtual machine running the Ubuntu operating system. The database, Prometheus tool, and Apache JMeter were installed and configured on the virtual machine, and the developed applications were deployed there. The specifications of the hardware used for testing are listed in Table 2.

Table 2: Test environment specification

Component	Specification
Virtual machine	VirtualBox 7.1.4
Operating system	Ubuntu 18.04.6 LTS
Processor	Intel Core i7-11800H @ 2.30GHz
Memory	24GB DDR4

The software configuration for the test environment is provided in Table 3.

Table 3: Software specification used during research

Tool	Version
Java	21.0.5
Spring Boot	3.3.3
GraphQL Spring Boot	16.9.0
Scala	3.5.0
Akka HTTP	10.5.3
Sangaria	4.1.1
Apache JMeter	5.6.3
Prometheus	2.55.1
H2 Engine	2.2.220

To vary the load on a single application thread during request processing, three data volumes were prepared. Each volume is characterized by a different number of records in the orders table. The number of records in the customers and products tables remained constant for all volumes. The data volumes and their characteristics are described in Table 4. To simulate a real-world application load, three groups of virtual users executing simultaneous requests were created. Characteristics of user groups is described in Table 5.

Table 4: Description of data volumes

S	Small data volume – 100 records
M	Medium data volume – 10 000 records
L	Large data volume – 300 000 records

Table 5: Description of user groups

S	Small users group – 10 users
M	Medium users group – 100 users
L	Large users group – 300 users

To analyze scenarios involving these variables, test scenarios were developed as detailed in Table 6. They are the combination of all data volume and users group sizes. This matrix allowed a comprehensive evaluation of the system's performance under diverse conditions, highlighting potential bottlenecks. Furthermore, this setup facilitated the identification of scalability issues that could arise when increasing data volumes or user loads.

Table 6: Description of test scenarios

Scenario	Data volume size	Users group size
S1	S	S
S2	S	M
S3	S	L
S4	M	S
S5	M	M
S6	M	L
S7	L	S
S8	L	M
S9	L	L

4. Results

The research is focused on measuring request processing times at the controller and method levels, as well as the memory usage and processor load.

The obtained results are presented in the form of tables. Their interpretation will be discussed in the "Discussion" chapter. The execution times at different levels provide a comprehensive understanding of applications performance under varying loads. Additionally, the collected data will enable an in-depth comparison of the efficiency of different algorithms and their impact on resource utilization. In each table, the observed anomalies are bolded. They will also be reviewed in the following chapter, ensuring a detailed analysis of performance metrics, enabling a clear identification of bottlenecks and inefficiencies.

Table 7 shows the execution times of programs at the algorithm level. For each scenario (S1-S9), algorithms A1, A2, and A3 were executed, with results described in each cell. Table 8 presents the times recorded in the programs at the controller level, encompassing data retrieval

from the database and subsequent processing via an algorithm method.

Table 7: Time obtained for Algorithms A1 – A3 on the method level

Scenario	Average time of execution [ms] for A1; A2; A3	
	Java	Scala
S1	4.76; 1.21; 0.72	2.77; **8.49**; 1.17
S2	27.12; 19.33; 0.93	2.12; 4.11; 0.65
S3	4.95; 2.73; 2.36	1.02; 2.09; 0.33
S4	47.65; 55.47; 29.46	20.18; 45.09; 16.39
S5	142.70; 40.19; 46.15	7.13; 10.83; 5.61
S6	37.33; 23.80; 17.48	4.21; 11.55; 6.07
S7	586.48; 117.76; 189.84	105.65; **295.82**; 108.35
S8	65.76; 87.02; 85.88	73.44; **259.96**; 163.63
S9	68.99; 84.57; 92.57	44.04; **182.14**; 86.94

Table 8: Time obtained for Algorithms A1 – A3 on the controller level

Scenario	Average time of execution [ms] for A1; A2; A3	
	Java	Scala
S1	**223**; 266; 243	288; 105; 215
S2	607; 571; 590	42; 64; 38
S3	786; 720; 777	27; 28; 25
S4	927; 1014; 1067	369; 382; 291
S5	4705; 3561; 3982	102; 118; 99
S6	4646; 3581; 3344	108; 122; 112
S7	5812; 5253; 5807	4434; 3901; 4828
S8	36696; 38264; 38917	3270; 3371; 3556
S9	56050; 70666; 58080	3304; 3362; 3218

Table 9 presents the maximum heap size used by the JVM during tests. Table 10 showcases the maximum processor usage. Each cell details resource consumption for Algorithms A1, A2, and A3.

Table 9: Maximum heap size measured for algorithms A1 – A3

Scenario	Maximum heap size [MB] for A1; A2; A3	
	Java	Scala
S1	63; 48; 51	35; 51; 37
S2	69; 76; 55	35; 49; 43
S3	68; 111; 112	83; 68; 100
S4	67; 66; 52	68; 48; **102**
S5	318; 368; 392	121; 230; 87
S6	1024; 1059; 763	422; 584; 470
S7	1962; 2262; 1392	1847; 2315; **2590**
S8	10144; 9032; 8576	6832; 8543; 8039
S9	10704; 14319; 12917	7648; 7728; 5929

Table 10: Maximum processor load measured for Algorithms A1 – A3

Scenario	Maximum processor load [%] for A1; A2; A3	
	Java	Scala
S1	42.5; 42.7; 51.4	25.2; 42.2; 38.2
S2	70.8; 51.7; 71.3	65.7; **57.5**; 70.5
S3	64.5; 83.9; 69.2	**68.4**; 65.0; 66.8
S4	100.0; 100.0; 100.0	70.9; 67.6; 85.8
S5	100.0; 100.0; 100.0	98.2; 100.0; 100.0
S6	100.0; 100.0; 100.0	83.1; 86.7; 91.2
S7	100.0; 100.0; 100.0	100.0; 100.0; 100.0
S8	100.0; 100.0; 100.0	100.0; 100.0; 100.0
S9	100.0; 100.0; 100.0	100.0; 100.0; 100.0

5. Discussion

The results collected in the previous chapter allow for the in-depth analysis, highlighting key findings, identifying patterns, and addressing observed anomalies. The implications of these findings are examined in the context of the research objectives and hypotheses.

In Table 7, it is evident that in most scenarios, Scala achieves significantly better results, with a lower average execution time for methods. However, in Scenario S1 and Algorithm A2, Scala processed data considerably slower

— by over 600%. Algorithm A2 also proved problematic for Scala in Scenarios S7, S8, and S9.

Table 8 highlights Scala and the Akka framework's clear dominance over Java and Spring. For a larger number of users, such as in Scenario S1 with Algorithm A1, Scala exhibited a performance advantage of up to 45 times. Only in Scenario S1 with Algorithm A1 Java processed requests faster by almost 30%. Interestingly, with Scala and Akka, an increasing number of users resulted in faster processing of individual requests, demonstrating the power and scalability of this framework. In contrast, Java with Spring required significantly more time for processing individual requests as the number of users increased.

Based on the data in Table 9, Scala with the Akka framework optimizes memory usage better, especially for larger data volumes. In the most demanding application scenario (S9), Algorithm A2 used over 14GB of memory in Java, whereas Scala did not exceed 8GB in the same case. Java used significantly less memory only in two cases – for Algorithm A3 in scenarios S4 and S7, respectively by approximately 50% and 45%. However these cases are not crucial, because mentioned scenarios concern the smallest user groups.

Table 10 further underscores Scala and Akka's dominance. Although Java with Spring occasionally used less processor for small data volumes, it consistently utilized 100% of available power for medium data volumes. In comparison, Scala only reached this level for two algorithms in Scenario S5. For large data volumes, both applications utilized 100% processor. The only cases where Java consumed less processor than Scala were scenario S2 with Algorithm A2 and scenario S3 with Algorithm A1. But in both cases the advantage is not higher than 10%, which is not significant as gains which Scala achieves in other cases.

Figures 2, 3, and 4 present program execution times at the method level using bar charts. These charts illustrate scenarios characterized by small (S1), medium (S5), and large (S9) workloads, respectively.

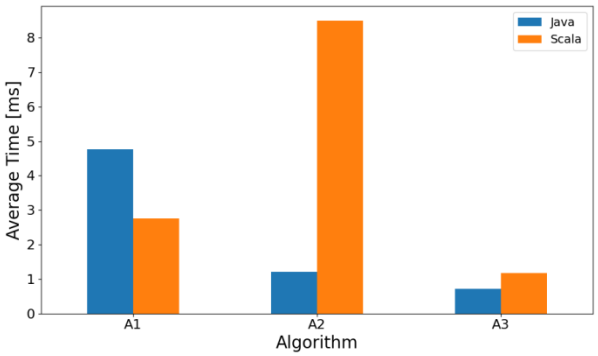

Figure 2: Average calculation times on method level for scenario S1 (10 users and 100 records).

Figure 2 shows how the performance of different languages varies depending on the data processing algorithm.

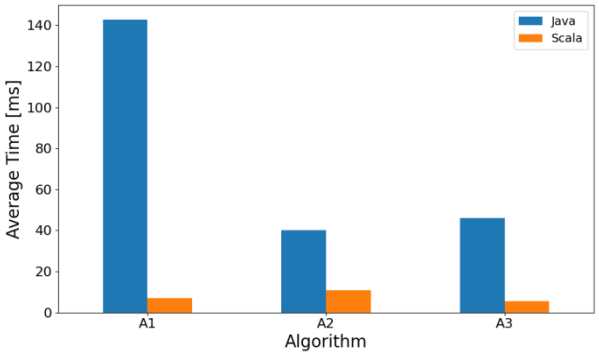

Figure 3: Average calculation times on method level for scenario S5 (100 users and 10 000 records).

Figure 3 highlights the significant dominance of Scala when processing medium-sized data volumes.

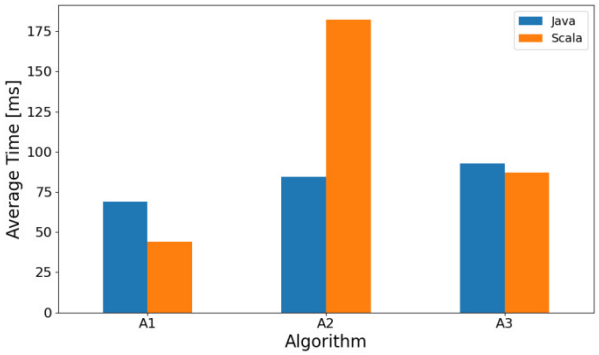

Figure 4: Average calculation times on method level for scenario S9 (300 users and 300 000 records).

Analysis of the chart in Figure 4 leads to conclusions similar to those drawn for small data volumes. When processing large data volumes, the algorithm used for data processing becomes a critical factor, making it challenging to definitively determine which language performs better.

Figures 5, 6, and 7 offer a graphical interpretation of program execution times at the GraphQL controller level. These figures, like those for methods, focus on scenarios with small, medium, and large workloads.

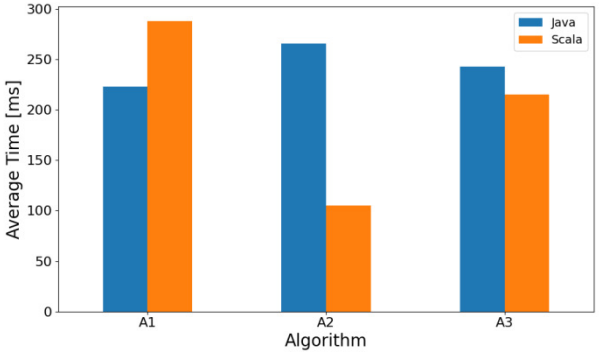

Figure 5: Average calculation times on controller level for scenario S1 (10 users and 100 records).

Figure 5 shows that for small workloads, it is difficult to determine which technology handles individual requests better.

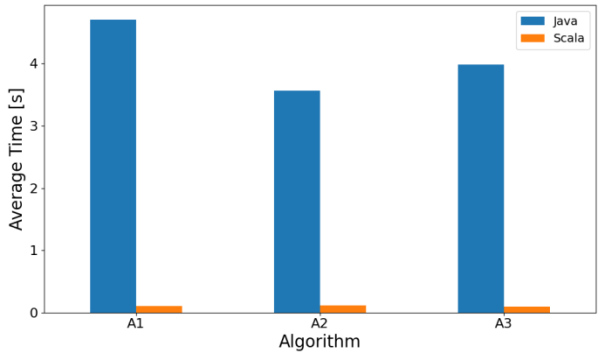

Figure 6: Average calculation times on controller level for scenario S5 (100 users and 10 000 records).

Figure 6 demonstrates the clear dominance of Scala and Akka over Java and Spring when processing requests under medium workloads. For example, for algorithm A1, Java performed a single request about 45 times longer than Scala.

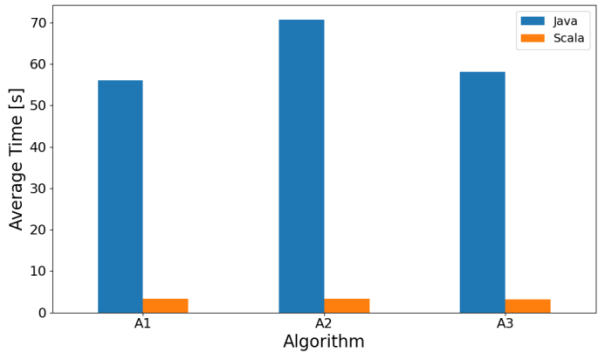

Figure 7: Average calculation times on controller level for scenario S9 (300 users and 300 000 records).

Figure 7 illustrates the superiority of the Scala application over Java for processing requests under heavy workloads. For algorithm A2, it took about 20 times longer for Java application to perform a single request comparing to Scala.

Figures 8 and 9 present the relationship between JVM heap memory usage and the number of users. They compare Algorithm A2 for medium and large data volumes.

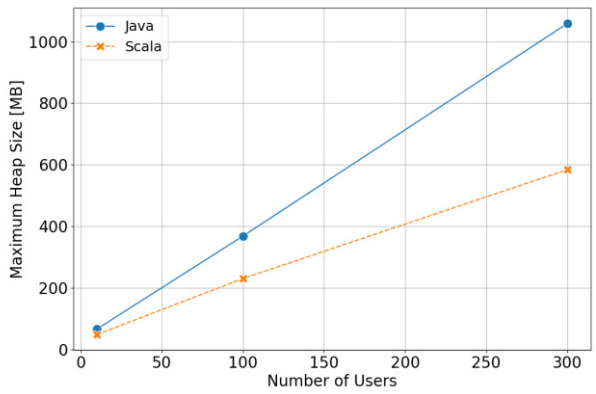

Figure 8: Relationship between number of users and heap size for Algorithm A2 and medium data volume.

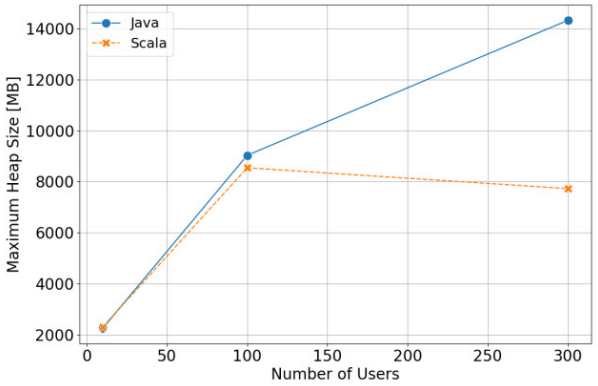

Figure 9: Relationship between number of users and heap size for Algorithm A2 and large data volume.

The charts in Figures 8 and 9 show Scala's advantage over Java in memory usage. Notably, for larger data volumes, this advantage becomes even more pronounced in Scala's favor.

Figures 10 and 11 compare processor usage during request processing for both web applications. Algorithm A1 was used for the comparison, focusing on small and medium data volumes.

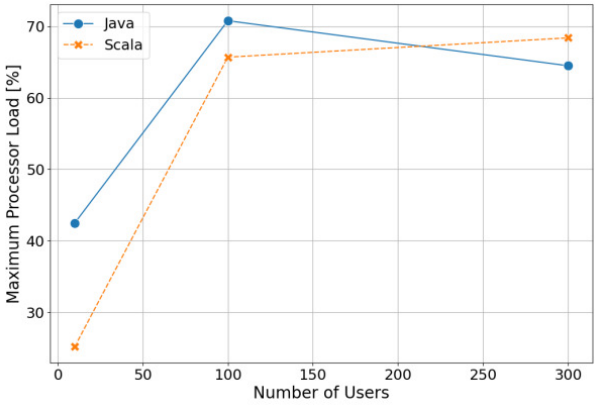

Figure 10: Relationship between number of users and processor load for Algorithm A1 and small data volume.

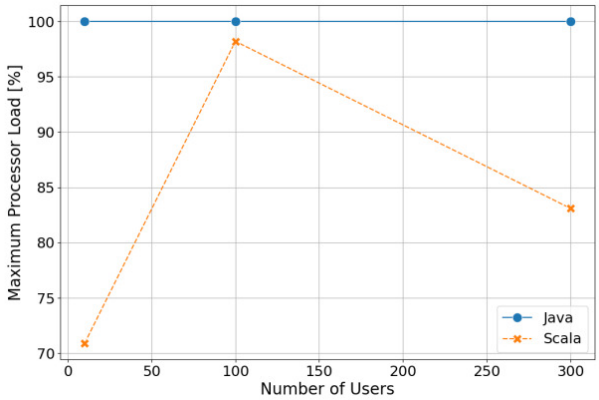

Figure 11: Relationship between number of users and processor load for Algorithm A1 and medium data volume.

The charts in Figures 10 and 11 again indicate Scala's superiority over Java. Although for small data volume the situation is less clear – Java used less processor for a large number of users, in every case for medium data volume,

Java consistently utilized 100% of processor resources, a result not observed for Scala.

The obtained results demonstrated that for a large data volume (scenarios S7, S8, S9), behaviour deviating from the trends observed in small and medium data volumes was noted. For example, the A1 algorithm implemented in Java in scenario S8 achieved better time performance at the method level than in scenario S5. However, in the case of the application written in Scala, resource utilization for the large data volume decreased between the medium and large user groups.

The analysis of reports generated using the JMeter tool after the tests revealed that applications processing a large data volume were unable to handle the load generated by simultaneous requests from multiple users. This resulted in timeout errors during the processing of HTTP requests. Based on these reports, a table was created showing the percentage of timeouts in the test scenarios for the large data volume (Table 11).

Table 11: Percentage of timeouts measured for large data volume

Number of users	Algorithm	Percentage of timeouts [%]	
		Java	Scala
10	A1	0	0
	A2	0	0
	A3	0	0
100	A1	25	44
	A2	0	52
	A3	0	20
300	A1	93	66
	A2	89	68
	A3	92	64

The presented summary indicates that for a small user group, no timeouts occurred. In scenarios with a medium user group, timeouts appeared in all algorithms in the Scala application and in the A1 algorithm of the Java application. For the large data volume, timeouts accounted for a significant percentage of all requests: on average, 91% in the Java application and 66% in the Scala application.

Figures 12, 13, and 14 present detailed data on the percentage of timeouts for algorithms A1, A2, and A3 depending on the number of users.

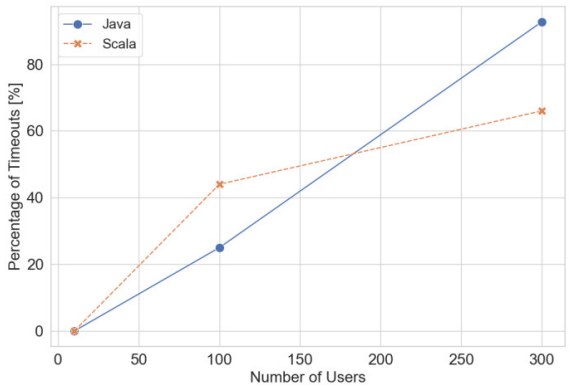

Figure 12: Relationship between number of users and percentage of timeouts for algorithm A1.

As shown in Figure 12, Java performed better under medium load; however, under heavy load, Scala gained the upper hand.

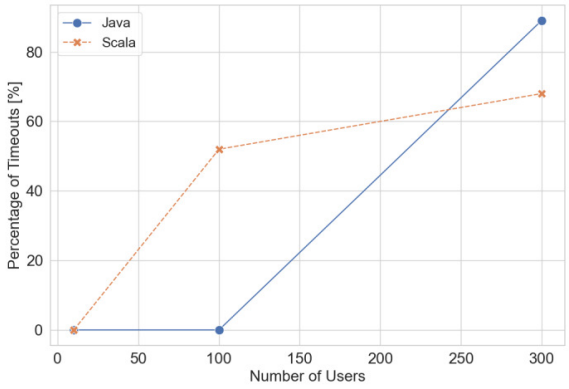

Figure 13: Relationship between number of users and percentage of timeouts for algorithm A2.

Figure 13 indicates that for small and medium user groups, the Java application processed all requests, but for the large user group, Scala demonstrated greater resilience.

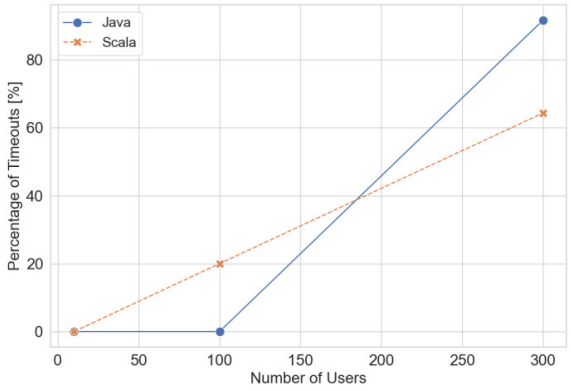

Figure 14: Relationship between number of users and percentage of timeouts for algorithm A3.

Based on figure 14, in algorithm A3 Java dominated in the case of small and medium data volumes, whereas under heavy load, Scala managed the load more effectively. The situation is similarly to algorithm A2.

Due to hardware limitations, both applications, regardless of implementation technology, struggled with processing large data volumes under high load. The conducted analysis indicates that timeouts are a significant factor affecting the performance evaluation of the applications. Nevertheless, it can also be concluded that in scenarios with medium load levels, Java performed better at processing data. Under high load, the Scala application proved to be more resilient compared to the Java application.

6. Conclusions

The aim of the study was to determine which language, Java or Scala, is more efficient in the context of data processing. For this purpose, the research encompassed several algorithms, data volumes of varying sizes, and different numbers of users performing requests.

The analysis of the results allowed the verification of the thesis and research hypotheses. Hypothesis H1 – "Scala processes both small and large data volumes faster" can be partially confirmed. In the vast majority of the scenarios examined, Scala processed larger data volumes faster, especially for larger data volumes, which are more significant from a business perspective. Hypothesis H2 – "Scala processes data faster using multithreading" must be confirmed. The results indicate that Scala performs better in multithreaded processing, as evidenced by the analysis of execution times at the GraphQL controller level. Hypothesis H3 – "Scala uses less memory during data processing" must be confirmed. The research results clearly show Scala as the technology better optimizing memory usage. Hypothesis H4 – "Scala uses less processor load during data processing" must also be confirmed. Processor usage analysis shows that Scala manages processor utilization significantly better. This advantage is particularly noticeable during the processing of medium-sized data volumes. Based on the verification of the presented hypotheses, the research thesis: "Scala is a more efficient solution for data processing in web applications" is partially confirmed by the research findings.

The collected research results led to the following conclusions:

1. It is not possible to definitively state which programming language processes data faster. Depending on the algorithm used for data processing, Java or Scala performs better in specific cases.
2. Scala and the Akka framework perform significantly better when processing requests from many users simultaneously compared to Java with Spring. The research observed instances where Scala with Akka processed requests several thousand percent faster.
3. Scala better optimizes memory usage during data processing. This advantage became particularly evident when processing larger data volumes.
4. Scala uses less processor. While both technologies utilized all available computing power for large data volumes, Scala demonstrated better processor utilization for smaller and medium-sized data volumes.

In summary, the results of the study indicate that Scala is an attractive technology suitable for developing various types of software, including web applications. The Akka framework deserves special attention due to its capabilities under high user loads. However, Java remains a solid choice due to its stability, extensive ecosystem of tools and libraries, and large developer community. Selecting the appropriate technology for software development requires in-depth analysis and consideration of multiple factors, including technical, business, and developer skill aspects.

References

[1] B. Eckel, Thinking in Java, Upper Saddle River, N.Y. Prentice Hall, Hoboken, 2014.

[2] TIOBE Index, https://www.tiobe.com/tiobe-index/, [22.11.2024].

[3] B. Evans, Java: The Legend, O'Reilly Media, Sebastopol, 2015.

[4] J. Engel, Programming for the Java Virtual Machine, Addison-Wesley, Boston, 1999.

[5] J. Swartz, Learning Scala, O'Reilly Media, Inc., Sebastopol, 2014.

[6] 10 Use Cases for Scala: What is Scala Used for?, https://www.sitech.me/blog/10-use-cases-for-scala-what-is-scala-used-for, [22.11.2024].

[7] D. Pariag, T. Brecht, A.S. Harji, P.A. Buhr, A. Shukla, D.R. Cheriton, Comparing the performance of web server architectures, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (2007) 231-243, https://doi.org/10.1145/1272996.1273021.

[8] D.R. Lakshmi, S.S. Mallika, A Review on Web Application Testing and its Current Research Directions, International Journal of Electrical and Computer Engineering 7 (2017) 2132-2141, http://doi.org/10.11591/ijece.v7i4.pp2132-2141.

[9] H.K. Dalla, A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS.NET Core, Journal of Physics: Conference Series 1933 (2021) 012041, http://doi.org/10.1088/1742-6596/1933/1/012041.

[10] K. Buszewicz, Performance analysis of languages working on Java Virtual Machine based on Java, Scala and Kotlin, Journal of Computer Sciences Institute 15 (2020) 189-195, https://doi.org/10.35784/jcsi.1609.

[11] A. Kołtun, B. Pańczyk, Comparative analysis of web application performance testing tools, Journal of Computer Sciences Institute 17 (2020) 351-357, https://doi.org/10.35784/jcsi.2209.

[12] Spring Framework Documentation, https://docs.spring.io/spring-framework/reference/, [22.11.2024].

[13] Akka Documentation, https://doc.akka.io/reference/, [22.11.2024].

[14] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Helion, Gliwice, 2014.

[15] M. Seabra, M.F. Nazário, G. Pinto, REST or GraphQL?: A Performance Comparative Study, Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (2019) 123-132, https://doi.org/10.1145/3357141.3357149.