Volume Special Issue, Issue 1, Page 49-60, 2024 https://doi.org/10.46810/tdfd.1425959 Research Article

BİNGÖL UNIVERSITY

Türk Doğa ve Fen Dergisi
Turkish Journal of Nature and Science
www.dergipark.gov.tr/tdfd

Performance Analysis of Compression Algorithms on Matrix Data: Data Transfer Optimization in Microservices Architectures

Faruk ATASOY[1*], Alper AKKAYA[1], Nadir KOCAKIR[1], Önder KARADEMİR[1]

[1] Özdilek Ev Tekstil San. ve Tic. AŞ, Özveri Ar-Ge Merkezi, Bursa, Türkiye
Faruk ATASOY ORCID No: 0009-0005-4177-9852
Alper AKKAYA ORCID No: 0009-0007-1927-6989
Nadir KOCAKIR ORCID No: 0000-0001-7421-0631
Önder KARADEMİR ORCID No: 0000-0001-5757-7335

**Corresponding author: faruk.atasoy@ozdilek.com.tr*

(Received: 26.01.2024, Accepted: 17.04.2024, Online Publication: 01.10.2024)

Keywords
Data Compression Algorithm,
Data Transfer,
Gzip,
Deflate,
Deflate,
Bz2

Abstract: With the rapid proliferation of microservices architectures these days, the efficient and fast transfer of large matrix data between services has become a significant challenge. This study presents an analysis aimed at finding solutions to this challenge. The analysis addresses the compression and decompression of large matrix data, focusing on lossless compression algorithms to optimize data transfer without data loss. The study is implemented on an example scenario. This scenario is taken from a project with a microservice architecture. In the example scenario, an image processing service developed in Python programming language generates 640x480 matrix data. After going through a compression algorithm, this data is periodically transferred to a backend service developed in C# programming language. This data is then stored in a database. In the final stage, decompression operations are performed so that this data can be used for reporting. The performance of various compression algorithms in the data compression, database storage and report generation stages is extensively tested. Within the scope of the study, tests were performed using five different compression algorithms (Gzip, Zlib, Deflate, Brotli and Bz2). The results are obtained through performance tests aimed at determining the most optimized end-to-end solution. Analyzing the performance of the compression algorithms on the example scenario, the Brotli algorithm gives the most optimal result in terms of both speed and compression size. This work makes an important contribution to data transfer optimization in microservice architectures and provides a reference for research in this area by presenting the performance analysis of various compression algorithms.

Veri Sıkıştırma Algoritmalarının Matris Verileri Üzerindeki Performans Analizi

Anahtar Kelimeler
Kayıpsız Veri Sıkıştırma Algoritmaları,
Veri Transferi,
Microservices,
Matris Verisi

Öz: Son zamanlarda mikroservis mimarilerinin hızla yayılmasıyla birlikte, büyük matris verilerinin hızlı ve verimli bir şekilde servisler arasında transferi, önemli bir zorluk haline gelmiştir. Bu çalışma, bu zorluğa çözüm bulmayı amaçlayan bir analiz sunmaktadır. Analiz, büyük matris verilerinin sıkıştırma ve açma işlemlerini ele almakta ve veri transferini optimize etmek için veri kaybı olmadan çalışan sıkıştırma algoritmalarına odaklanmaktadır. Çalışma, bir örnek senaryo üzerinde uygulanmıştır. Bu senaryo, mikroservis mimarisine sahip bir projeden alınmıştır. Örnek senaryoda, Python programlama dili ile geliştirilmiş bir görüntü işleme servisi, 640x480 boyutunda bir matris verisi üretmektedir. Bu veri, bir sıkıştırma algoritmasından geçtikten sonra periyodik olarak C# programlama dili ile geliştirilmiş bir back-end servise transfer edilmektedir. Bu veri daha sonra bir veritabanında depolanmaktadır. Son aşamada, bu verinin raporlama için kullanılabilmesi için açma işlemleri gerçekleştirilmektedir. Çeşitli sıkıştırma algoritmalarının performansı, veri sıkıştırma, veritabanı depolama ve rapor oluşturma aşamalarında detaylı bir şekilde test edilmiştir. Çalışma kapsamında beş farklı sıkıştırma algoritması (Gzip, Zlib, Deflate, Brotli ve Bz2) kullanılarak testler gerçekleştirilmiştir. Sonuçlar, en optimize edilmiş end-to-end çözümü belirlemeye yönelik performans testleri ile elde edilmiştir. Örnek senaryo üzerinde sıkıştırma algoritmalarının performansını analiz ederken, Brotli algoritması hem hız hem de sıkıştırma boyutu açısından en optimal sonucu vermektedir. Bu çalışma, mikroservis mimarilerinde veri transferi optimizasyonuna önemli bir katkı sağlamakta ve çeşitli sıkıştırma algoritmalarının performans analizini sunarak bu alandaki araştırmalara referans oluşturmaktadır.

1. INTRODUCTION

The rapid evolution of information technology is replacing traditional monolithic structures with microservice architectures that seek more agile and scalable systems. By decomposing software applications into small, independent services, microservice architectures aim to accelerate development processes, provide scalability and increase flexibility.

While the rise of microservice architectures has made software applications more flexible and scalable, this transformation has also brought with it the need to efficiently transfer large data sets. The fast and efficient transfer of large data sets between services has become critical. Problems such as bandwidth consumption, latency and storage costs arise during the transfer of big data. These problems are optimized through a number of methods such as data compression and coding, protocol optimization, cache utilization, parallel processing, distributed storage, etc.

This paper aims to optimize the problems that arise during the transfer of large matrix data between microservices by using compression algorithms. It focuses on the performance of Gzip, Zlib, Bz2, Deflate and Brotli lossless compression algorithms in the compression and decompression processes of matrix data transfer between Python and C# based applications with a message broker in between. It examines data transfer at each step of the process and provides detailed performance analysis. It also discusses their performance on matrix variants with different types of data sets. The results were obtained in the context of a real-life project.
In addition, this study aims to shed light on the optimization of other microservices projects with large data transfers in different data types by using compression algorithms.

The rest of the paper is organized as follows: Section 2 summarizes the literature review. Section 3 discusses the problem definition, information about compression algorithms, information about the algorithms that will be tested to solve the problem, and the types of matrices used in the testing. Section 4 presents the results of the performance tests. Section 5 contains the evaluation and comments on the final results.

2. LITERATURE REVIEW

With the rapid deployment of microservice architectures, the fast and efficient transfer of large matrix data between services emerged as a major challenge. In this context, a survey of the existing literature was conducted to gain various perspectives. Research on topics such as microservice architecture, lossless compression algorithms, matrix data, inter-service communication, etc., guided this review. Specifically, the realization of a gap in the literature concerning the optimization of large matrix data transmission between microservices motivated the investigation to fill this void. Enliçay et al. [1] undertook a study on the optimization of large datasets between microservices using the Deflate and Gzip algorithms. It was observed that no significant differences existed between the two algorithms, leading to the continued preference for the more prominent Gzip algorithm. Öztürk et al. [2] conducted tests on various NoSQL database technologies utilizing LZ4 and Zlib algorithms, finding that the Zlib algorithm, when used with MongoDB, achieved the best compression ratio. Conversely, when Snappy was employed, LevelDB yielded the fastest compression results. Deorowicz [3] provided a comprehensive examination of globally utilized compression algorithms, elucidating the differences and usage directions of these algorithms, along with underlying strategies and methods. This comparison facilitated an understanding of the appropriateness of each algorithm based on different scenarios.

Ramu [4] asserted that the adoption of microservice architecture has led to significant advancements in building scalable and flexible software systems. The study examined and evaluated the impact of microservices architecture on performance, focusing on vital elements such as inter-service communication, service discovery, data management, fault tolerance, and scalability. The findings contributed to the understanding of microservices and offered practical recommendations for architects and developers to optimize the performance of their applications. Kodituwakku and Amarasinghe [5] investigated and compared the performance of lossless data compression algorithms, evaluating their efficacy in compressing text data. The entropy of these algorithms was discussed, and an experimental comparison was conducted. Tapia et al. [6] described the transition of a project developed in monolithic architecture to the microservice level, noting the absence of mention regarding data optimization. Somashekar [7] presented effective solutions for optimizing data across microservices, focusing on optimizing the setup and layout of systems rather than the data itself. Semunigus and Pattanaik [8] analyzed differences between Huffman encoder, LZW encoder, and Arithmetic encoder, which was deemed insufficient for addressing both the analysis of matrix defects and the control of foundation disassembly. The blocks suitable for the project were identified as the high-level changes constructed with these blocks.

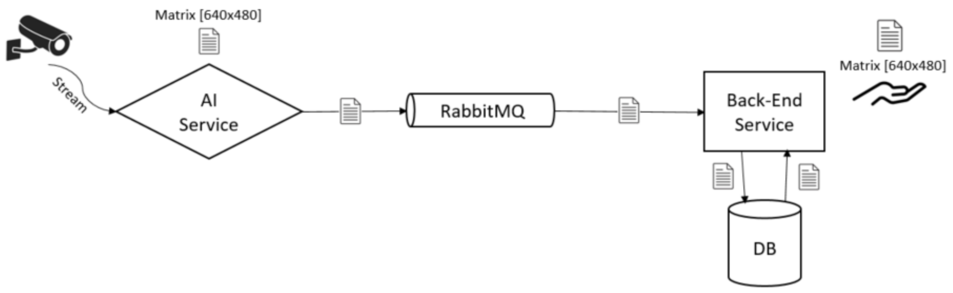

Figure 1. The journey of matrix data on the project architecture.

The mentioned studies above provide a comprehensive overview of matrix data transfer in microservices architectures. However, it is crucial to consider the limitations and gaps in the existing research in this area. This article focuses on existing studies to understand the shortcomings in the current literature and highlight how its own research aims to contribute to filling these gaps.

3. LITERATURE REVIEW

3.1. Problem Description

The system we are working on is built within the microservices architecture. The system architecture is shown in Figure 1. The end user completes the setup of the artificial intelligence service by selecting one of the previously registered cameras in the system. In the subsequent process, the Angular-based front-end communicates the necessary configuration to the .NET Core-based back-end service through APIs. After the writing process to the database, the same configurations are stored on Redis, a discrete cache mechanism, in a key-value format. Simultaneously, the service responsible for coordinating Python-based artificial intelligence services is notified using Redis's pub-sub mechanism. Once the coordinating service is notified, it retrieves the settings from the relevant Redis key and initiates the process of the heat map module based on these settings. The artificial intelligence service now monitors camera recordings via the stream URL of the relevant camera within the specified time period in the settings. At the end of the period, it publishes the movement data of detected individuals to the relevant queue in RabbitMQ, which serves as a message broker. Another C#-based back-end service, listening to this queue, receives the relevant report and writes it to the PostgreSQL database.

The journey of the data necessitating performance optimization within the architecture is addressed in Figure 1. The artificial intelligence service monitors the camera, sends the acquired data to the back-end service through RabbitMQ. Here, the data is later written to the database for use during report preparation.

All these operations are conducted to monitor, analyze, and generate reports on the movements of people visiting social areas, primarily malls and hypermarkets. The obtained data is stored in the database for later reporting. If desired, data sets are presented to end users on a daily, monthly, or yearly basis. The visualization of these data sets benefits from the heat map method.

The periodic data obtained by the artificial intelligence service is of a two-dimensional integer array type. Therefore, we can refer to it as a two-dimensional matrix. The first dimension of this matrix has a size of 640, and the second dimension has a size of 480 characters. The majority of matrix data consists of 0 parameters. Other parameters different from 0 represent the number of people passing through that point. The reason for the matrix size being 640x480 is due to the camera resolution. Due to the excess of matrix elements, the size of the data sometimes reaches megabytes. The largeness of the data has made it essential to optimize the process in terms of delays in transfer, storage costs, and resource consumption. Especially in the preparation of heat map reports, pulling, summing, and serving matrix data of 640x480, which can be tens or hundreds, from the database will result in significant resource consumption and delays.

Research has concluded that utilizing compression algorithms would be beneficial to optimize the process. Figure 2 shows that the data needs to be optimized at 4 different points.

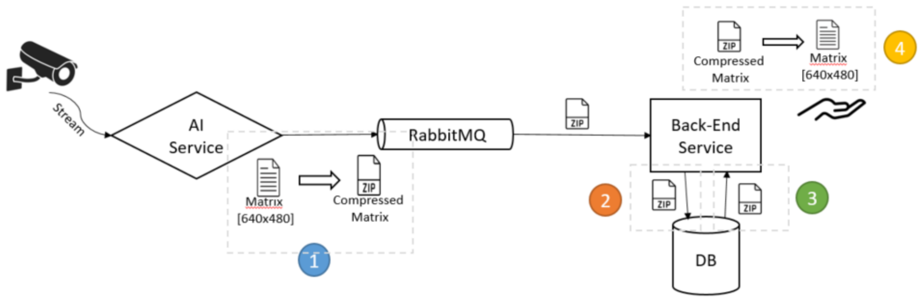

Figure 2. Showing the stages where matrix data is processed on the architecture.

3.2. Compression Algorithms

A data compression algorithm is a software or hardware method used to process data more efficiently during storage or transmission. Data compression algorithms are typically divided into two main categories. [10]

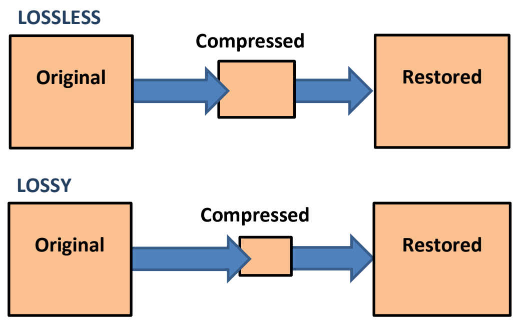

Figure 3. Behavior of compression algorithms.

The first consideration when determining algorithms to be considered for evaluation is to prefer lossless compression algorithms, as each parameter in the matrix is deemed to be highly significant. The second condition is to select algorithms that can be used in both the C# and Python programming languages. After making this distinction, the final condition to consider is the active preference and usage of algorithms in the software industry. Through literature reviews and research, we have decided to evaluate five different algorithms. Five compression algorithms (Gzip, Zlib, Deflate, Brotli, and Bz2) have been examined for data transfer optimization in micro services architectures.

3.2.1. Gzip

Gzip [11] is an algorithm developed for data compression purposes. It primarily operates using the Lempel-Ziv (LZ77) compression algorithm and Huffman coding. LZ77 analyzes the data and provides compression by identifying consecutive repeating patterns. Huffman coding adds an extra compression step by providing a shorter representation for more frequently occurring symbols. The main features of Gzip include:

• Effective Compression: It efficiently compresses data, reducing file sizes effectively.
• LZ77 and Huffman Coding: It performs compression using LZ77 and Huffman coding as fundamental algorithms.
• Usage in Web Pages: It is commonly used, especially on web servers, to compress text-based data sent to browsers. This enables faster loading of web pages.

Gzip is a widely used compression algorithm across various applications and is preferred for tasks such as data transfer over the internet and file archiving.

3.2.2. Zlib

Zlib [9] is a data compression library developed based on the Deflate algorithm. Zlib is commonly used for compression and archiving operations in the "gzip" format, although Zlib itself only encompasses compression features. The key features of Zlib include:

• Fast and Efficient Compression: Zlib reduces file sizes by compressing data quickly and efficiently.
• Portability: Zlib can be used on many different platforms and operating systems, allowing for a wide range of applications.
• Ease of Use: The Zlib library provides an easy-to-use API, facilitating the integration of compression operations for developers.

Zlib is widely utilized for data compression and archiving by web browsers, servers, databases, and many other applications.

3.2.3. Deflate

Deflate [13] is an effective algorithm used for data compression, combining two fundamental components: the LZ77 (Lempel-Ziv 1977) compression algorithm and Huffman coding.

• LZ77 (Lempel-Ziv 1977): This algorithm analyzes the data and provides compression by identifying consecutive repeating patterns. It encodes new blocks by referencing previous data blocks.
• Huffman Coding: It ensures symbols in the compressed data are represented with shorter codes. Frequently used symbols have shorter codes, while less frequently used symbols have longer codes.

The Deflate algorithm is widely used for compression and decompression operations. File formats like Gzip, Zip, and PNG can utilize the Deflate compression algorithm. Deflate is known for its fast operation, high compression ratios, and broad usability. Web browsers, file archiving, databases, and many applications prefer Deflate to optimize data transfer.

3.2.4. Brotli

Brotli [9] is a data compression algorithm developed by Google. It is designed to replace other popular compression algorithms and is specifically used to speed up the loading of web pages. Brotli aims to achieve higher compression ratios and better performance than Deflate algorithms (such as gzip and zlib). Some of its features include:

•High Compression Ratios: Brotli provides high compression ratios, especially for text-based content. This helps in delivering web pages faster and more efficiently.
• Adaptive Algorithm: Brotli uses an adaptive algorithm that can adjust the compression strategy based on the type and structure of the content. This results in more effective compression for various data types.
• Support for Web Browsers and Servers: Brotli is supported by modern web browsers and web servers. This enables faster loading of web pages, enhancing the overall user experience.

• Platform-Independent: Brotli can be used on different platforms and operating systems, offering a wide range of usability.

The use of Brotli plays a significant role, particularly in data transfer over the internet and web performance optimization. Faster loading of web pages contributes to an improved user experience.

3.2.5. Bz2

Bz2 is a compression algorithm and accompanying program used for data compression, also known as "bzip2." It was developed by Julian Seward. Unlike traditional compression algorithms like ZIP and Gzip, bz2 combines techniques such as Burrows-Wheeler Transform (BWT) and Huffman coding. Some of its features include:

• High Compression Ratios: Bz2 typically provides higher compression ratios compared to some other compression algorithms.
• Burrows-Wheeler Transform (BWT): This transformation reorganizes the data, highlighting repeated patterns and leading to more effective compression.
• Huffman Coding: It ensures symbols in the compressed data are represented with shorter codes. This involves shorter codes for frequently used symbols and longer codes for less frequently used symbols.
• Platform-Independent: Bz2 can be used on different platforms and operating systems.
• Data Archiving: It is commonly used for file archiving and distribution.
The bz2 compression algorithm is widely used, especially in UNIX and Linux-based systems, and users often prefer the command-line tool named "bzip2" for compression tasks.

3.3. Matrix Types Examined

The performances of these algorithms have been measured on three different large-sized matrix data sets in processes such as compression, compressing and writing to the database, compressing, writing to the database, and then reading, as well as compressing, writing to the database, reading again, and decompressing. Although the type of matrix that solves our problem is a matrix containing a large number of zeros, to shed light on and assist in future studies related to the subject, we did not limit our tests to a single type of matrix but conducted tests on various matrix types. The general format of the examined large-sized matrix data is as follows:

• **Random Matrix**
Elements are randomly chosen integers. Example: [{1896322487, 423, 837, 29895, 5, ...}, {284997874, 5713597144, 77, 6852, ...}, ...]

• **Sequential Elements Matrix**
Elements are consecutively repeating integers. Example: [{11, 11, 22, 11, 11, 22, ...}, {11, 11, 22, 11, 11, 22, ...}, ...]

• **Matrix with Many Zeros**
Matrix with mostly zero elements, and integers. Example: [{0, 0, 598, 0, 0, 0, 85479, ...}, {0, 0, 8, 0, 0, 923559, 0, ...}, ...]

4. EXPERIMENTAL RESULTS AND DISCUSSION

Within the scope of the study, five different compression algorithms (Gzip, Zlib, Deflate, Brotli, and Bz2) underwent performance testing. These tests were focused on data compression ratios as well as compression and decompression times. All operations were performed according to the hardware and software standards shared in Table 1.

Table 1. Test Environment Components.

Sistem	Windows 11 (10.0.22000.2295/21H2/SunValley) 12th Gen Intel Core i7-1260P, 1 CPU, 16 logical and 12 physical cores, Kingston SSD snv2s2000g
.NET SDK	7.0.400
Database	PostgreSQL 15.3, compiled by Visual C++ build 1914, 64-bit
Measurement Tool	BenchmarkDotNet v0.13.7

4.1. Experimental Results

The numbered figures in Figure 4 represent the operations that will be subjected to performance testing. The location of these operations in the architecture can be found in Figure 2 in Section 3.1. The first numbered figure represents compressing the matrix, the second numbered figure represents writing the compressed matrix to the database, the third figure represents reading the compressed matrix data from the database, and the

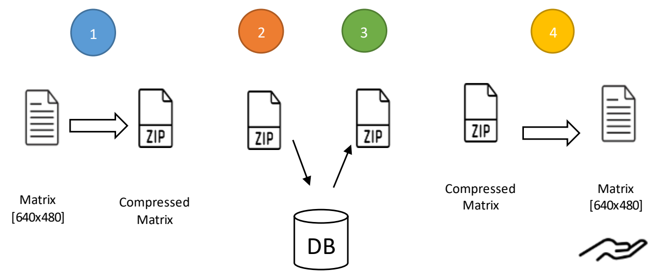

Figure 4. Operations included in the test and their numbers.

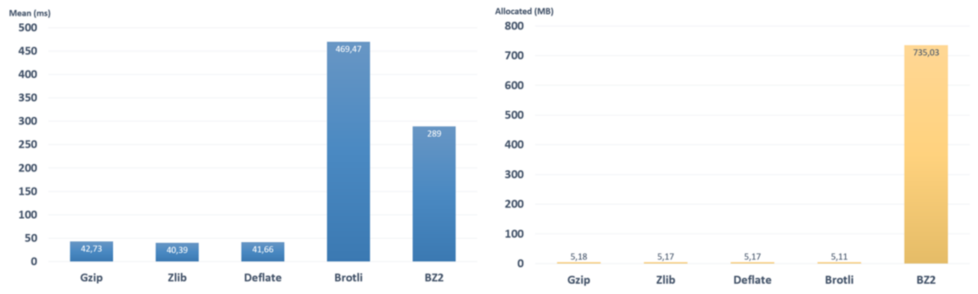

Figure 5. Complex matrix performance measurements belong to just first operation in Figure 4.

fourth and last figure represents restoring the compressed matrix data.

4.1.1. Complex matrix performance measurements

Graphs in Figure 5 compare the performance of five different compression algorithms in terms of two key metrics: processing speed and memory usage. The first graph shows the average processing time of each algorithm in milliseconds, showing that Gzip, Zlib and Deflate have similar and relatively low processing times. In contrast, Brotli's processing time is higher than all three, while BZ2 has by far the slowest processing time.

The second graph compares memory consumption in megabytes. Here again, Gzip, Zlib and Deflate show similar and quite low values in terms of memory consumption, while Brotli uses slightly more memory than these three, but the BZ2 algorithm requires a much larger amount. These results emphasize the need for an in-depth evaluation of compression processes in terms of time and memory efficiency.

These data suggest that the choice of compression algorithm should be carefully tailored to the application requirements. For example, it can be concluded that Gzip or Zlib may be preferable when speed is critical, while Brotli or BZ2 may be more appropriate in scenarios where compression ratio is more important. However, the high memory consumption and low speed of BZ2 can be a significant disadvantage, especially for real-time systems or resource-constrained environments.

Graphs in Figure 6; The first graph depicts the mean time taken by each algorithm to compress a dataset, measured in milliseconds (ms). Here, Gzip, Zlib, and Deflate show relatively similar and modest compression times at 60.44 ms, 46.65 ms, and 46.28 ms respectively, suggesting efficient performance in time-sensitive applications. Brotli, however, takes substantially longer at 498.63 ms, which may be a trade-off for better compression ratios. BZ2 is significantly slower at 182.75 ms, indicating that while it may offer high compression ratios, it is less suitable for scenarios where time efficiency is critical.

The second graph compares the algorithms based on the amount of memory allocated during compression, measured in megabytes (MB). Gzip, Zlib, and Deflate again cluster closely together, with each requiring just over 5 MB of memory, pointing towards a low memory footprint. Brotli's memory allocation is marginally higher at 5.11 MB, which could be justified by its potentially better compression efficiency. However, BZ2 stands out with a substantially higher memory requirement of 735.03 MB, which is orders of magnitude greater than the others. This suggests that BZ2's compression technique, while perhaps yielding high compression ratios, is extremely memory-intensive, making it less feasible for systems with limited memory resources.

In summary, when considering a compression algorithm for practical use, it's crucial to weigh the trade-offs between compression time, memory usage, and compression efficiency. Gzip, Zlib, and Deflate present themselves as balanced choices for general purposes. In contrast, Brotli may be more suitable when compression efficiency outweighs the need for speed, and BZ2 might only be practical when memory resources are abundant, and compression ratio is the paramount concern.

Graphs in Figure 7; In the first graph, we examine the mean compression time in milliseconds (ms) for each algorithm. Gzip shows a mean time of 72.91 ms, Zlib at 50.21 ms, and Deflate at 49.12 ms, which are fairly close

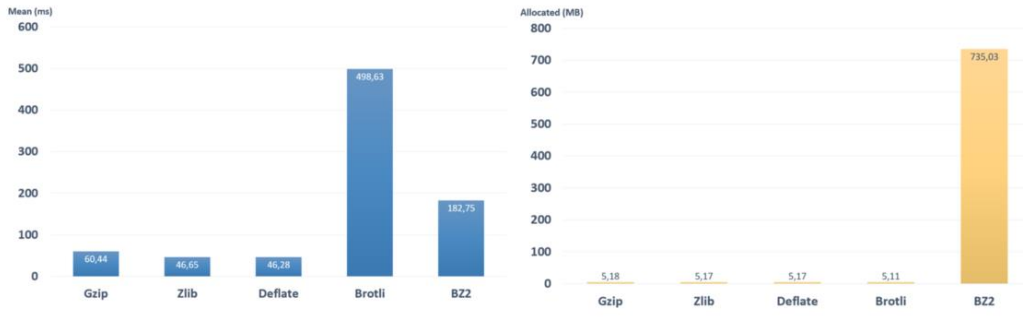

Figure 6. Complex matrix performance measurements belong to first and second operations in Figure 4.

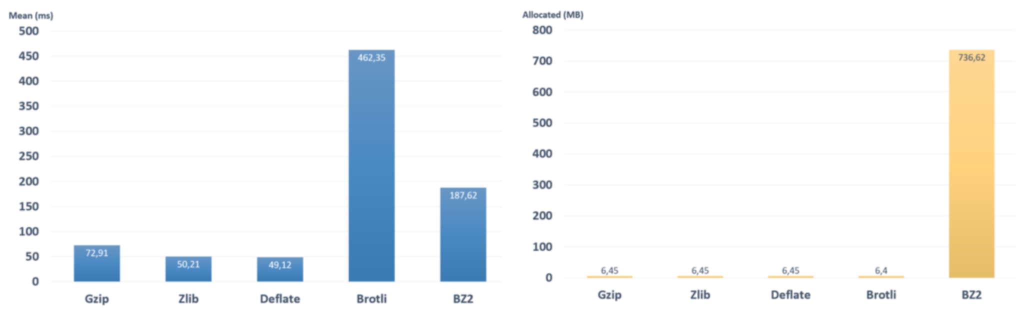

Figure 7. Complex matrix performance measurements belong to first, second and third operations in Figure 4.

in performance, indicating they could be suitable for tasks where moderate compression speed is required without significant time constraints. Brotli dramatically increases to 462.35 ms, suggesting a possible preference for a higher compression ratio at the cost of time efficiency. BZ2 is notably quicker than Brotli at 187.62 ms, but still substantially slower compared to Gzip, Zlib, and Deflate, which may make it less ideal for time-critical applications.

The second graph shows memory allocation during the compression process in megabytes (MB). Gzip, Zlib, and Deflate maintain a low memory footprint at approximately 6.45 MB each, making them practical for environments with memory usage constraints. Brotli has a similar memory requirement at 6.4 MB, slightly less than the others, which is impressive considering its compression time. BZ2, however, requires a staggering 736.62 MB of memory, which is an order of magnitude higher than its counterparts. This high memory demand could severely limit BZ2's usability, especially in systems where memory is a scarce resource.

In conclusion, while Gzip, Zlib, and Deflate offer balanced performance with low memory consumption, Brotli might be considered when the compression ratio is more important, provided that the longer compression time is acceptable. BZ2, despite its moderate compression time, may only be viable in systems where ample memory is available and the highest possible compression ratio justifies its significant memory allocation. These insights are critical when selecting an algorithm for specific use cases, particularly when balancing the need for speed, efficiency, and available system resources.

Graphs in Figure 8; The first graph details the mean compression time, revealing that Gzip, Zlib, and Deflate take 2,422.40 ms, 2,693.30 ms, and 499.9 ms respectively. These times suggest that while Gzip and Zlib have slower compression rates, Deflate is significantly faster, almost five to six times quicker than its counterparts. Brotli and BZ2 offer even better performance at 354.2 ms and 335.8 ms, respectively, which might make them preferable in time-sensitive scenarios where efficiency is paramount.

In the second graph, memory allocation is depicted, with Gzip, Zlib, and Deflate all showing minimal differences in their memory usage, ranging between 7.63 MB to 7.62 MB. Brotli's allocation is marginally lower at 7.57 MB, which could be considered negligible in most cases. However, BZ2 demonstrates a substantial increase in memory requirements, utilizing 742.29 MB. This considerable memory usage implies that BZ2 might only be suitable in situations where memory resources are abundant and a high compression ratio is required, despite its good performance in terms of compression speed.

Overall, the graphs suggest that while Brotli and BZ2 offer the best compression times, the latter does so at a significantly higher memory cost. This data is vital when considering the operational context of these algorithms, as it highlights the need to balance compression speed with resource consumption according to specific application needs.

4.1.2. Sequential elements matrix performance measurements

Graphs in Figure 9; In the first graph, the mean compression times are relatively low for Gzip, Zlib, and Deflate, recorded at 24.46 ms, 22.99 ms, and 22.62 ms, respectively. These figures suggest a high level of efficiency, with little to differentiate between the three in

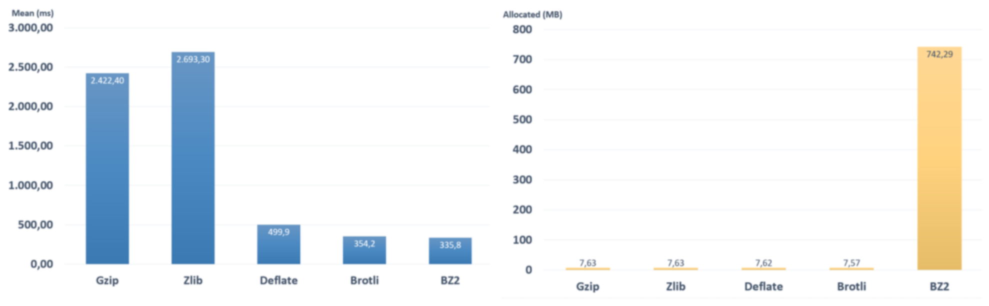

Figure 8. Complex matrix performance measurements belong to first, second, third and fourth operations in Figure 4.

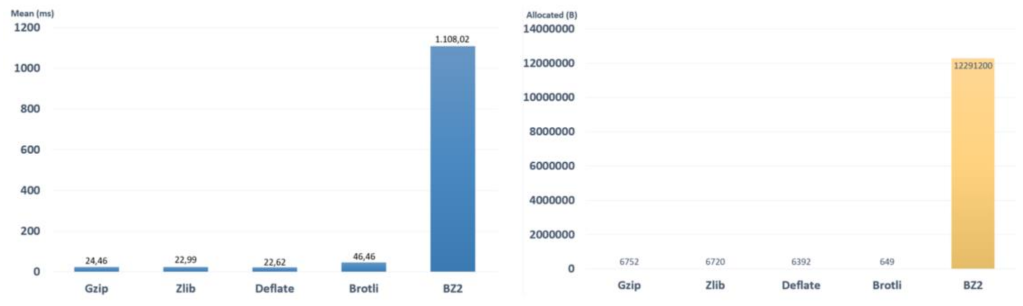

Figure 9. Sequential elements matrix performance measurements belong to just first operation in Figure 4.

terms of speed. Brotli shows a higher time of 46.46 ms, which could be attributed to its more complex compression algorithms designed for better compression ratios. BZ2, however, shows a significantly higher mean time of 1,108.02 ms, indicating that it is considerably slower than its counterparts. This would likely make BZ2 less suitable for applications where time efficiency is critical.

The second graph is particularly revealing regarding memory allocation. Gzip, Zlib, and Deflate use similar and minimal amounts of memory, with allocations at 6752 bytes, 6760 bytes, and 6392 bytes, respectively, which demonstrates their suitability for memory-constrained environments. Brotli has a slightly higher allocation at 649 bytes, which remains relatively modest. However, BZ2's memory allocation is an outlier, requiring a colossal 12,291,200 bytes. This is orders of magnitude greater than the other algorithms, suggesting it may be impractical for most applications due to its high memory demands, despite any potential benefits in compression ratio.

In summary, these visual data points emphasize the importance of considering both compression speed and memory usage when selecting an algorithm. While Gzip, Zlib, and Deflate offer the best balance for general use, Brotli might be an alternative for specific use cases that can tolerate slightly longer compression times. BZ2's high resource consumption makes it suitable only for niche applications where its compression benefits outweigh the significant memory requirements.

Graphs in Figure 10; From the first graph, we observe that Gzip, Zlib, and Deflate have quite competitive mean compression times of 25.76 ms, 23.58 ms, and 23.3 ms respectively. These small differences are unlikely to impact the choice of algorithm in scenarios where compression time is somewhat flexible. Brotli's mean time is slightly higher at 34.87 ms, which is still within an acceptable range for many applications, considering that it typically achieves better compression ratios. BZ2, however, has a mean time of 1,153.63 ms, which is substantially higher than the others. This may limit its use to non-time-sensitive processes where compression efficiency is more critical.

In the second graph, memory allocation in kilobytes (KB) is represented, showing that Gzip, Zlib, and Deflate all have modest memory requirements at 65.13 KB, 65.1 KB, and 64.78 KB, respectively. Brotli is slightly more efficient at 59.28 KB. In stark contrast, BZ2 requires a dramatically higher amount of memory at 12,063.71 KB, which is nearly 200 times greater than its nearest competitor. This high memory allocation suggests that BZ2's use cases might be very specialized, where the benefits of its compression outweigh the cost in terms of memory usage.

The data from these graphs suggest that for most general purposes, Gzip, Zlib, and Deflate offer a good balance of speed and memory efficiency. Brotli stands out as a strong candidate when slightly higher compression times are acceptable, and memory efficiency is a priority. BZ2 appears to be a specialized tool that might be reserved for unique situations where its heavy memory use can be justified, likely in environments where memory is not a constraint and maximum compression is the primary goal.

Figure 11; From the compression time perspective, Gzip and Zlib are the quickest, with times of 26.51 ms and 21.89 ms, respectively, making them suitable for applications where speed is essential. Deflate and Brotli exhibit slightly longer compression times at 24.7 ms and 35.29 ms, respectively, with Brotli's longer time possibly

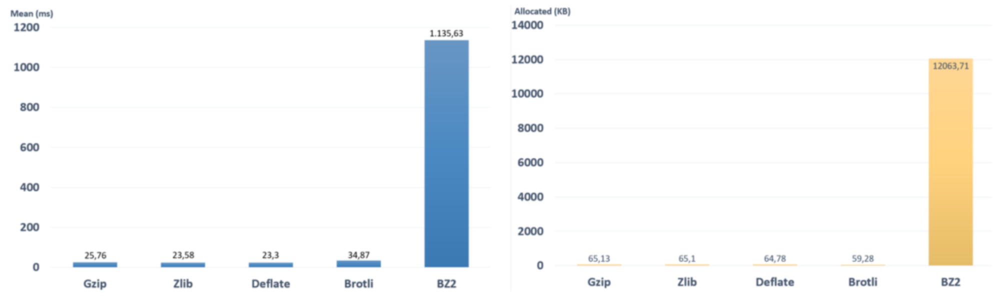

Figure 10. Sequential elements matrix performance measurements.

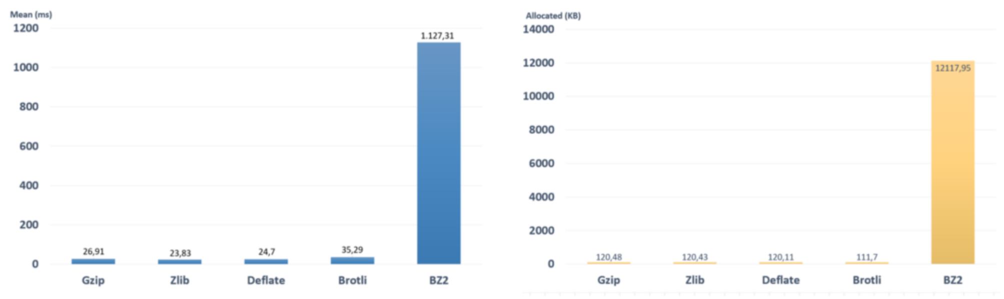

Figure 11. Sequential elements matrix performance measurements belong to first, second and third operations in Figure 4.

reflecting its focus on achieving higher compression ratios. BZ2 is the outlier, taking considerably longer at 1,127.31 ms, suggesting it might be less suited for time-sensitive processes.

The memory allocation chart shows that Gzip, Zlib, Deflate, and Brotli have similar and relatively low memory footprints, all under 1.3 MB. This is beneficial in resource-constrained environments where maintaining a small memory usage is crucial. In stark contrast, BZ2 requires a substantially larger amount of memory, at 17.27 MB, which may hinder its practicality in systems with limited memory availability.

Figure 12; Gzip and Zlib demonstrate relatively moderate compression times at 1,842.34 ms and 1,975.51 ms, respectively. Deflate follows closely, showing a time of 69.66 ms, suggesting a faster performance that could be beneficial in time-sensitive applications. Brotli shows a compression time of 1,502.17 ms, which is slower compared to Gzip and Zlib but faster than BZ2, which has the highest mean time at 11,879.34 ms, indicating it may be less efficient for rapid compression needs.

When examining memory allocation, Gzip, Zlib, Deflate, and Brotli are quite efficient, each requiring slightly over 1 MB of memory. This low memory footprint is advantageous for environments where resource conservation is essential. However, BZ2's memory requirement is much higher at 17.36 MB, making it potentially impractical for memory-constrained systems despite its slower compression time.

In an academic context, these findings would suggest that while Brotli and BZ2 have slower compression times, they might offer better compression ratios, which could be a trade-off depending on the application's requirements. However, the significantly higher memory usage of BZ2 could limit its use to systems where memory resources are not a concern.

4.1.3. Matrix With Many Zero Performance Measurements

Graphs in Figure 13; For compression time, Gzip is the fastest at 34.46 ms, closely followed by Zlib and Deflate at 29.32 ms and 30.61 ms respectively. Brotli is slightly slower at 56.54 ms, while BZ2 is the slowest at 67.27 ms.

Memory allocation shows Gzip, Zlib, and Deflate using less than 4 KB of memory, indicating high efficiency. Brotli requires slightly more at 671 bytes. BZ2, however, requires a much larger memory size of 12,197,576 bytes, which is substantially more than its counterparts.

In summary, Gzip, Zlib, and Deflate are efficient in both time and memory usage, suitable for most applications. Brotli trades off some speed for compression quality, while BZ2, due to its high memory demand, may only be practical for specific use cases where its compression benefits outweigh its resource usage.

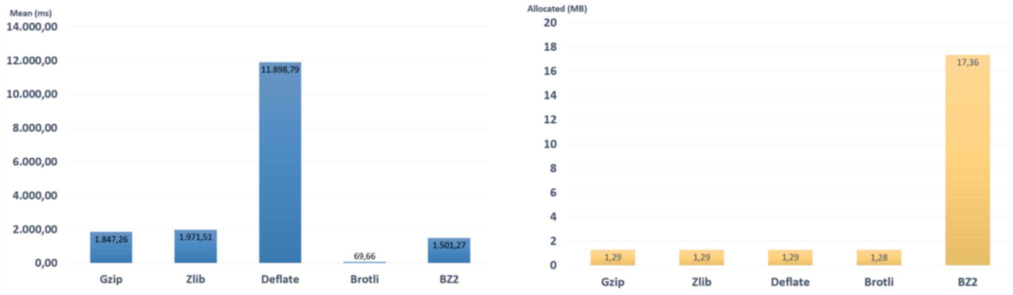

Figure 12. Sequential elements matrix performance measurements belong to first, second, third and fourth operations in Figure 4.

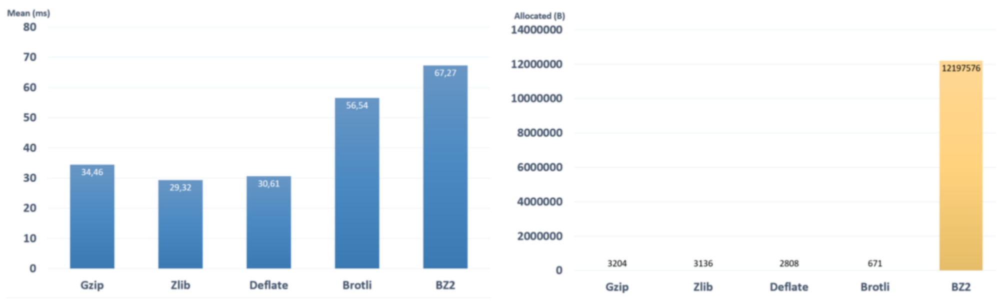

Figure 13. 4.2.3. Matrix with many zero performance measurements belong to just first operation in Figure 4.

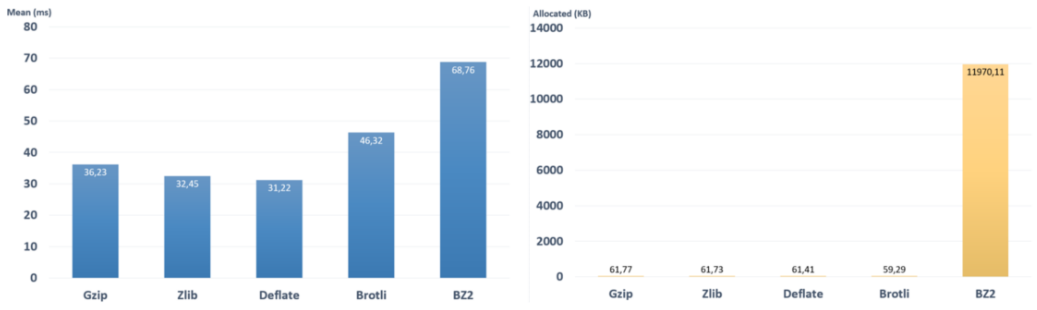

Figure 14. Matrix with many zero performance measurements belong to first and second operations in Figure 4.

Graphs in Figure 14; The mean compression times are fairly close for Gzip, Zlib, and Deflate, recorded at 36.23 ms, 32.45 ms, and 31.22 ms respectively, which suggests these algorithms are quite efficient. Brotli shows a higher time at 46.32 ms, and BZ2 is the slowest at 68.76 ms, which could be a drawback for rapid compression needs.

On the memory allocation front, Gzip, Zlib, and Deflate are comparable, requiring between 61.77 KB to 61.41 KB. Brotli is slightly more efficient at 59.29 KB. However, BZ2's memory requirement is exceptionally high at 11,970.11 KB, which may render it less practical for environments with limited memory resources.

In essence, Gzip, Zlib, and Deflate are suitable for general use, offering a good balance of speed and low memory usage. Brotli stands out as a slightly less efficient option in terms of speed but still maintains low memory use. BZ2, while slowest, may offer better compression ratios at the cost of significantly higher memory usage.

Graphs in Figure 15; Gzip has a compression time of 37.33 ms, Zlib is faster at 26.32 ms, and Deflate is very close to Zlib at 32.96 ms. Brotli is slower at 55.22 ms, and BZ2 is the slowest at 71.6 ms.

For memory usage, Gzip, Zlib, and Deflate are nearly identical, ranging from 115.23 MB to 115.02 MB. Brotli is marginally more efficient at 111.79 MB. BZ2's memory allocation is significantly higher at 12,023.52 MB, suggesting it is less efficient in terms of memory usage.

Overall, Zlib and Deflate are the most efficient in both time and memory usage, while BZ2's high memory allocation could limit its practicality despite its compression capability.

Graphs in Figure 16; In terms of compression time, Gzip, Zlib, and Deflate exhibit high performance, with mean times of 9,452.95 ms, 11,166.46 ms, and 10,679.48 ms, respectively, indicating they are quite fast. Brotli has a lower time at 575.5 ms, and BZ2 is the most time-efficient at 62.98 ms.

Memory allocation shows that Gzip, Zlib, Deflate, and Brotli have similar low memory usage, around 1.29 MB to 1.28 MB. In stark contrast, BZ2 requires significantly more memory at 17.27 MB.

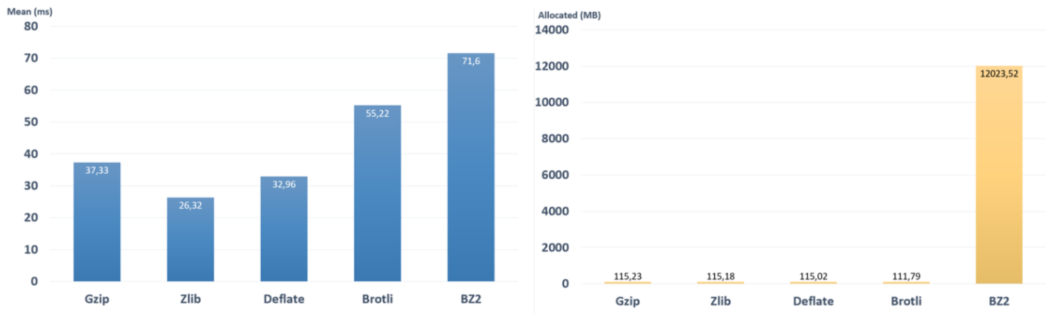

Figure 15. Matrix with many zero performance measurements belong to first, second and third operations in Figure 4.

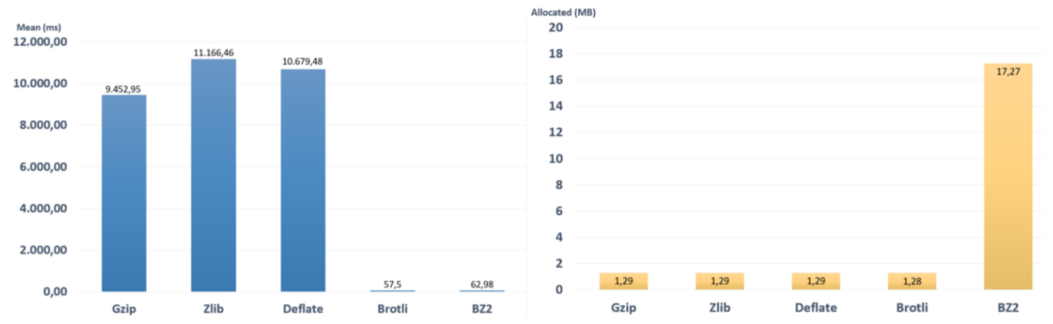

Figure 16. Matrix with many zero performance measurements belong to first, second, third and fourth operations in Figure 4.

BZ2 stands out with the lowest compression time but requires more memory, while Brotli offers a balance between efficient compression time and low memory usage. Gzip, Zlib, and Deflate, while slower, also maintain a low memory footprint.

4.2. Compression Performance Measurements

Figure 17 covers the performance of compression algorithms in minimizing the 63,684-byte matrix. As observed, the Brotli algorithm demonstrates a clear superiority compared to the others. Brotli, being the overall winner in performance tests, owes its distinction to its high compression ratio.

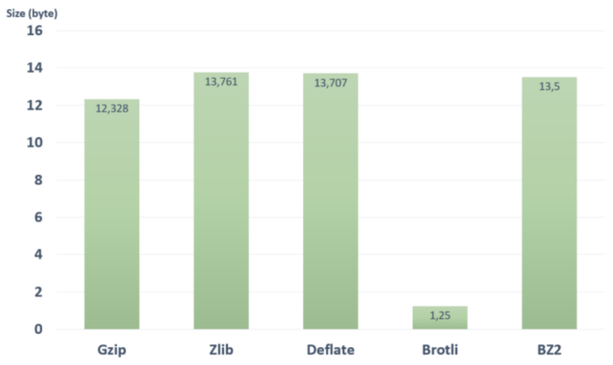

Figure 17. Compression ratios of algorithms.

5. DISCUSSION

The results of the experimental studies have shown that the Brotli algorithm provides the best performance. Brotli offers both a high compression ratio and fast compression and decompression times. Other algorithms either have a low compression ratio or slow processing times. These results provide a framework for the effective transfer of large matrix data in microservices architectures. An overall evaluation of the obtained results is presented below:

• Gzip, Zlib, and Deflate have been the most performant in the first three periods, providing similar results to each other.
• Brotli, despite lagging behind in performance in the first three measurements, has been the most efficient algorithm in the end-to-end test.
• While the BZ2 algorithm exhibited the worst performance in the first three periods, it yielded good results in the later periods; however, it did not meet the expected memory usage.
• The ultimate reason for Brotli being the optimal algorithm is its high compression ratio compared to others.
• Brotli algorithm provided the most optimal results for all types of matrices.

6. CONCLUSION

This study focuses on optimizing data transfer in microservices architectures and evaluates the performance of compression and decompression processes for large matrix data. The results obtained indicate that the Brotli algorithm provides the best solution in this context. It has been the most performant algorithm for three different types of matrices subjected to testing. This study serves as a reference for researchers and industry professionals interested in data transfer in microservices architectures. The provided information should not be limited to matrix-specific evaluations but should also be considered for other data formats.

Looking ahead, further research can expand upon this work by exploring a broader range of data formats such as text, JSON, and XML to determine the most effective compression algorithms for diverse data types. Additionally, investigating the distribution of compression and decompression tasks in distributed systems could offer insights into enhancing performance for large-scale data transfers. Embracing adaptive compression techniques and exploring hardware acceleration are promising directions that could lead to significant improvements in real-time data processing and efficiency. These future endeavors will build on the foundation laid by this study, offering a comprehensive understanding of data optimization in microservices architectures across various contexts and technologies.

Acknowledgement

This study was presented as an oral presentation at the "6th International Conference on Life and Engineering Sciences (ICOLES 2023)" conference.

REFERENCES

[1] Enliçay M, Şahin Ö, Ülger İ, Balçiçek ÖE, Baydarman MV, Taşdemir Ş. Veri Sıkıştırma Algoritmalarının Karşılaştırılması: Katılım Bankası Örneği. Konya: Selçuk Üniversitesi; 2014.

[2] Öztürk E, Mesut A, Diri B. The performance analysis of data compression algorithms used in NoSQL databases. In: Proceedings of the International Conference on Computer Science and Engineering (UBMK 2016); 2016 Oct; Tekirdağ.

[3] Deorowicz S. Universal lossless data compression algorithms [dissertation]. Gliwice: Silesian University of Technology; 2003.

[4] Ramu V. Performance impact of microservices architecture. Rev Contemp Sci Acad Stud. 2023;3.

[5] Kodituwakku SR, Amarasinghe US. Comparison of lossless data compression algorithms for text data. Indian J Comput Sci Eng. 2010;1(4):416-25.

[6] Tapia F, et al. From monolithic systems to microservices: a comparative study of performance. Appl Sci. 2020;10(17):5797.

[7] Somashekar G. Performance management of large-scale microservices applications [dissertation]. Stony Brook (NY): Stony Brook University; 2023.

[8] Semunigus W, Balachandra P. Analysis for lossless data compression algorithms for low bandwidth networks. J Phys Conf Ser. 2021;1964(4).

[9] Alakuijala J, et al. Brotli: A general-purpose data compressor. ACM Trans Inf Syst. 2018;37(1):1-30.

[10] Bulut F. Huffman algoritmasıyla kayıpsız hızlı metin sıkıştırma. El-Cezeri. 2016;3(2).

[11] Gailly J-l, Adler M. gzip [Internet]. Available from: http://ftp.gnu.org/gnu/gzip/gzip-1.6.tar.gz .

[12] Zlib Compression Library [Internet]. Available from: http://www.zlib.net/ [cited 2024 Jan 10].

[13] Deutsch P. Deflate compressed data format specification version 1.3. RFC 1951 (Informational). IETF; 1996.