Assessing the Efficiency of Java Virtual Threads in Database-Driven Server Applications

L. Lasić *, D. Beronić *, B. Mihaljević * and A. Radovan *
* Rochester Institute of Technology Croatia (RIT Croatia), Zagreb, Croatia
luka.lasic@mail.rit.edu, dora.beronic@mail.rit.edu,
branko.mihaljevic@croatia.rit.edu, aleksander.radovan@croatia.rit.edu

2024 47th MIPRO ICT and Electronics Convention (MIPRO) | 979-8-3503-8250-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/MIPRO60963.2024.10569754

Abstract — **Virtual Threads represent a contemporary structured concurrency model in Java Virtual Machine (JVM) seeking to increase the performance of multi-threaded Java applications by optimizing the utilization of the operating system (OS) resources. Virtual Threads were first introduced within the OpenJDK project Loom as lightweight threads based on the native implementation of thread schedulers inside the JVM that are less reliant on OS schedulers. Within the Java Development Kit (JDK), Virtual Threads were presented as a preview feature in JDK 19/20 and became fully implemented as a part of the standard JDK 21.**

Given the paucity of research on the efficiency of Java's Virtual Threads, particularly in cloud-based environments, we explored the role of Virtual Threads in enhancing Java's concurrency capabilities in the realm of database-driven cloud computing, particularly server applications with thread-per-request model and various backend databases. Our research examined Virtual Threads' efficiency in comparison with Java's traditional threads within database-driven server applications with use cases utilizing common data frameworks to access relational and non-relational databases. From our findings and preliminary results, we propose the possible utilization of Virtual Threads in modern database-driven framework-based server applications in the cloud.

Keywords - Virtual Threads, Java Virtual Machine, Thread Scheduling, Structured Concurrency, Java, Database Framework

I. INTRODUCTION

Database-driven web applications heavily depend on the dynamic exchange of data, as well as the efficacy of the underlying database systems responsible for managing the application-specific data. Such applications predominantly adhere to a client-server model wherein client interactions create numerous requests to trigger database actions encompassing data retrieval, modification, insertion, or deletion activities.

The performance of these applications is intrinsically tied to the efficiency of server-side technologies, such as Java, .NET Core, and PHP, in processing these requests as well as their utilization of hardware resources, mainly CPU and memory, during intensive blocking operations such as connecting or writing data to the database.

Commonly, a relatively complex process in software development is identifying issues related to thread scheduling, system calls, or I/O operations that significantly impact the application performance. For example, the complexity of the Java Virtual Machine (JVM) requires an in-depth comprehension of the internal JVM processes in improving application responsiveness, making it a complex task for software developers due to the scarcity of quality performance analysis tools capable of collecting data across various layers of the application [1].

Moreover, applications facing a high number of user interactions or managing complex database schemas with extensive datasets encounter significant challenges in executing query requests, particularly during the blocking processes for updating and inserting data records. In larger applications, this causes heavy loads on the server side and the database system by running frequent requests within limited time intervals, retrieving large amounts of data in a single request, or running intricate queries requiring extended processing times. These conditions often result in significant surges in response times as well as elevated consumption of CPU and memory resources, underscoring the imperative for the efficient utilization of operating system resources crucial for achieving scalability in database-driven applications.

This research delves into the exploration and analytical comparison of the efficiency of multi-threaded database-driven web applications in Java based on the thread-per-request model, used as benchmarks in combination with some of the most popular relational database systems (MySQL, PostgreSQL, and Oracle) and non-relational database systems (MongoDB, Neo4j, and Cassandra). Furthermore, based on the previous research [2,3], it focuses on the comparison of Java's traditional threads and Virtual Threads (VTs), a contemporary concurrency construct in Java presented within OpenJDK project Loom and fully implemented in the latest version 21 of Java Development Kit (OpenJDK JDK 21.0.2).

II. RELATED WORK

The burgeoning volume of digital information necessitates enhanced processing capabilities and efficiency from both applications and the underlying Database Management Systems (DBMS) tasked with managing extensive datasets. Therefore, evaluations of various applications and test cases using different DBMSs become indispensable for optimizing overall system performance tailored to specific use cases and application requirements.

An analysis of high-performance DBMSs based on TPC-H test queries conducted on PostgreSQL, MySQL, and Microsoft SQL Server [4], presented interesting findings.

Based on the averages of 10 TPC-H query runs with generated 10, 30 and 50 GB of data, MS SQL Server DBMS achieved the best performance in handling operations involving substantial data volume, while MySQL DBMS was identified as the least performant, attributed to its inability to optimally parallelize query execution across multiple CPU cores, a critical factor in optimizing computational efficiency.

Further exploration into the performance of CRUD (Create, Read, Update, Delete) operations unveiled contrasting efficiencies between relational and non-relational DBMSs. The comparative study executed within two analogous Java applications based on MySQL and CouchDB scrutinized varying data volumes and query complexities [5]. Findings suggest that MySQL, particularly in document-based data management, demonstrated heightened efficacy in managing operations on large datasets. Despite this, MySQL experienced a degradation in performance with increasing data volumes, particularly evident in SELECT operations. Nonetheless, it maintained a performance edge over CouchDB under conditions of lesser data, highlighting the nuanced performance landscape across different DBMS architectures and operational contexts.

In the examination of Java framework performance, specifically the Spring Data JPA framework across leading relational database management systems such as MySQL, PostgreSQL, Oracle, and Microsoft SQL Server in performing CRUD operations, presented that pairing with MySQL seems most promising, especially for DELETE operations. Further analysis reveals that PostgreSQL exhibits a marginally enhanced performance for creating and reading operations, and SQL Server had the best performance for reading operations [6].

Another study performed on the Hibernate framework measured its compatibility with several relational database management systems, including MySQL, PostgreSQL, Oracle SQL, and Microsoft SQL Server, when performing CRUD operations on tables with entry counts ranging from 1000 to 500 000 [7]. The findings indicate that SQL Server in combination with Hibernate exhibits superior READ operation performance. Conversely, its efficacy diminishes significantly for DELETE operations. PostgreSQL demonstrated optimal performance for databases housing fewer than 5000 entries, whereas Oracle's performance was more favorable in scenarios involving a larger number of entities, suggesting its suitability for database systems with extensive datasets.

Furthermore, an older empirical study of Java database frameworks, leveraging input from 3707 public GitHub Java projects, illuminated the prevalence of Java Database Connectivity (JDBC) and frameworks such as Spring, JPA, and Hibernate [8]. Notably, simpler JDBC was utilized in over 56% of all projects despite its lack of database schema abstractions. Its widespread adoption can be attributed to low-level database query utilization, allowing developers to execute complex SQL queries commonly unfeasible within higher-level database frameworks. Given the potential evolution of these trends, our research adopts JDBC as the foundation for assessing the compatibility of diverse database systems with Java concurrency constructs, sidestepping the variable performance metrics associated with different database frameworks.

These insights underscore the complex interplay between various Java frameworks and different DBMSs in executing CRUD operations on varying datasets, providing a need for a nuanced understanding of system performance. Since we wanted to examine the efficiency of database-driven applications with different database systems regarding contemporary Java concurrency constructs and traditional threads, we decided to use plain JDBC for our research without relying on the potential influence of a combination of Java database frameworks and selected DBMS on our results.

III. Concurency in Java

One of the key factors in enhancing the scalability of server-side applications lies in the concurrency capabilities of the programming languages used for their development. This capability is gauged by the volume of concurrent requests an application can manage within a given timeframe. In scenarios where applications engage in numerous blocking operations, such as input-output tasks or database write operations, concurrent process execution becomes less efficient. This inefficiency occurs as concurrent processes must await the completion of preceding blocked operations before initiating new ones, a precaution essential to averting erroneous behavior or producing unpredictable results.

A. Threads

The creation of Java threads significantly leans on the operating system (OS) kernel threads, making them inherently heavyweight. This relationship is encapsulated within the java.lang.Thread API, which acts as a slender wrapper interface around the OS threads. Consequently, the efficiency of Java threads is closely tied to the efficiency of the operating system scheduler, which is not platform-specific and accommodates various programming languages. This general-purpose approach is suboptimal for handling blocking operations like I/O or networking operations, as it necessitates that both the application thread and the corresponding OS thread must wait until the release of the lock on the resource.

Furthermore, due to the generic implementation, the OS lacks insight into the specific memory stack requirements of newly created Java threads, often leading to the allocation of unnecessarily larger memory stacks and contributing to the memory-intensive nature of creating Java threads.

To resolve this problem, some studies suggest the adoption of thread pools as a strategy to evade the creation of new threads for each new request and circumvent the overhead. The implementation of this model on the Producer-Customer Problem demonstrated an increase in performance in regard to response time [9]. This model allows the application to reuse already created threads, which are returned to the thread pool for future use after completing their tasks. However, this approach does not address the inherent inefficiencies related to thread scheduling within the Java Virtual Machine (JVM), including the switching between OS threads and Java's threads as well as the costly thread creation process,

occurring because of JVMs' heavy reliance on OS schedulers that are not language-specific.

B. Virtual Threads

Virtual Threads represent an innovative concurrency construct in Java, offering an alternative to the traditional java.lang.Thread model based on the native implementation of schedulers managed by the JVM.

This approach introduces a lightweight framework for executing concurrent operations, fundamentally comprising two core components:

1. Continuation – represents the current state of a running thread, which can be both captured and modified, thus allowing for the pausing and resuming of thread execution in a controlled manner.
2. Java Scheduler – maps the Virtual Thread to the carrier thread, facilitating the execution of specific tasks as required.

The creation of a Virtual Thread establishes an association with a continuation object, effectively encapsulating the thread's state. When a Virtual Thread engages in a blocking operation or requires a suspension, its continuation state is preserved, and the underlying carrier thread is released for reuse by another virtual thread. This process ensures that, following the completion of an input/output operation, the Virtual Thread can resume its activities by restoring its continuation object.

In this Thread implementation, the JVM is responsible for allocating and managing OS resources, dynamically mapping Virtual Threads to OS Threads for task execution. When the Virtual Thread gets blocked, the JVM releases the OS Thread, thereby allowing it to be used by another Virtual Thread instead of idly waiting for a resource to be unblocked.

This mechanism not only enhances application scalability by preventing OS Threads from remaining dormant during blockages but also potentially optimizes the utilization of platform threads. Consequently, this could lead to improved application performance, marking a significant advancement in the efficient management of concurrency within the Java ecosystem.

IV. Database Connectivity

Database connection pooling is a technique that enhances the efficiency of cloud-based Java applications by facilitating multi-threading interactions with the database. This technique maintains a pool of preconnected database connection objects within a connection pool, ready for use by application threads. Upon requiring database access, a thread retrieves an existing connection object from the pool, utilizes it for the necessary database operations, and subsequently returns it to the pool for future use by another thread. This connection pool manages the lifecycle of database connections, from their creation to termination, thereby mitigating the overhead associated with the repetitive opening and closing of connections and conserving time and system resources [10]. Hence, optimizing thread management and reusing database connections can significantly improve the performance of applications heavy on database operations.

This test case utilizes a selection of widely adopted databases spanning relational and non-relational systems. MySQL, PostgreSQL, and Oracle were selected for evaluation of relational database-based application performance using Structured Query Language (SQL). Among these, MySQL is reputed for its efficiency in read-intensive operations, PostgreSQL is recognized for its versatility across varying-size projects, and Oracle is esteemed for its robustness in large-scale enterprise systems. The impact of databases on the research outcomes may be influenced by the relatively small size of the dataset and the magnitude of requests employed during testing, highlighting the necessity to consider the database's operational strengths and the context of the application's requirements in performance evaluation.

The test case also includes industry-prevalent non-relational databases MongoDB, Cassandra, and Neo4j. MongoDB is a popular document-oriented NoSQL database, where data is stored in BSON (Binary JSON) format, which enhances the database's flexibility and scalability. This format is advantageous for applications requiring frequent read and write operations on document-like data, although it may exhibit performance degradation under intense workloads. Nonetheless, this limitation is not anticipated to impact the outcomes of this research [11]. Neo4j is a graph-type database designed for storage, querying, and management of data characterized by dense network-like interconnections. It employs nodes to facilitate the efficient handling of complex queries. However, it is pertinent to acknowledge that our test case does not leverage the database's inherent strengths in graph data processing, as the tasks undertaken are not inherently graph-oriented. Cassandra is a NoSQL database adhering to a wide-column store model, where data is partitioned across multiple in a cluster. This organization allows for data to be arranged in tables and columns, optimizing the database for high-volume write operations and achieving low latency in data retrieval [12].

V. Preliminary Results

A. Testing Environment

The experimental testing environment uses the Runtime Environment of OpenJDK 64-bit Server VM installed on macOS Monterey Version 12.7.2 operating system with 16GB 1600 MHz DDR3 memory and 2.8 GHz Quad Core Intel Core I7 processor. For the compilation, we used the latest long-term support (LTS) JDK 21, which introduced Virtual Threads as a finalized feature (previously introduced as a preview in the JDK 19 and a second preview in the JDK 20. The RESTful service test environment was combined with MySQL version 8.0.30, PostgreSQL 14.10, and Oracle 23c representing the relational databases, and Cassandra 4.1.3, Neo4j 5.16.0, and MongoDB 7.0.3 build 2 for NoSQL databases. Connectivity with the relational databases was established using the Apache Commons Database Connectivity (DBCP) or JDBC version 2.11.0. Proprietary Java drivers facilitated connections to Neo4j (version 5.4.0) and MongoDB (version 4.11.1) and Cassandra used the Java driver core 4.17.0. The databases were populated with the

same amount of data, consisting of 120 entries in a single table. The database connection pool size was set to the default number of connections (commonly 8 or 10) for the specific combination of the database and driver used.

To evaluate the performance with test cases across different database systems, two versions of a test application with an HTTP server based on RESTful services were developed to handle incoming requests based on a thread-per-request model - one utilized traditional threading and the other employed Virtual Threads. Both versions of the test application were connected with each database system utilizing Java JDBC for connections to MySQL, Oracle, and PostgreSQL, and database-specific drivers for MongoDB, Cassandra, and Neo4j databases. Performance testing was conducted using Vegeta, a load-testing HTTP-request-based tool, simulating an attack rate of 300 requests per second over 20-second spans.

Testing methodology encompassed executing each CRUD operation across 3 sequential runs, with the initial 2 serving as warm-up sessions to precondition the system. The CRUD functions were invoked through targeted HTTP requests against the HTTP server, including GET, POST, PUT, and DELETE operations. The same process was used on both implementations of the testing application with HTTP server, handling requests with Threads and Virtual Threads across all databases under review.

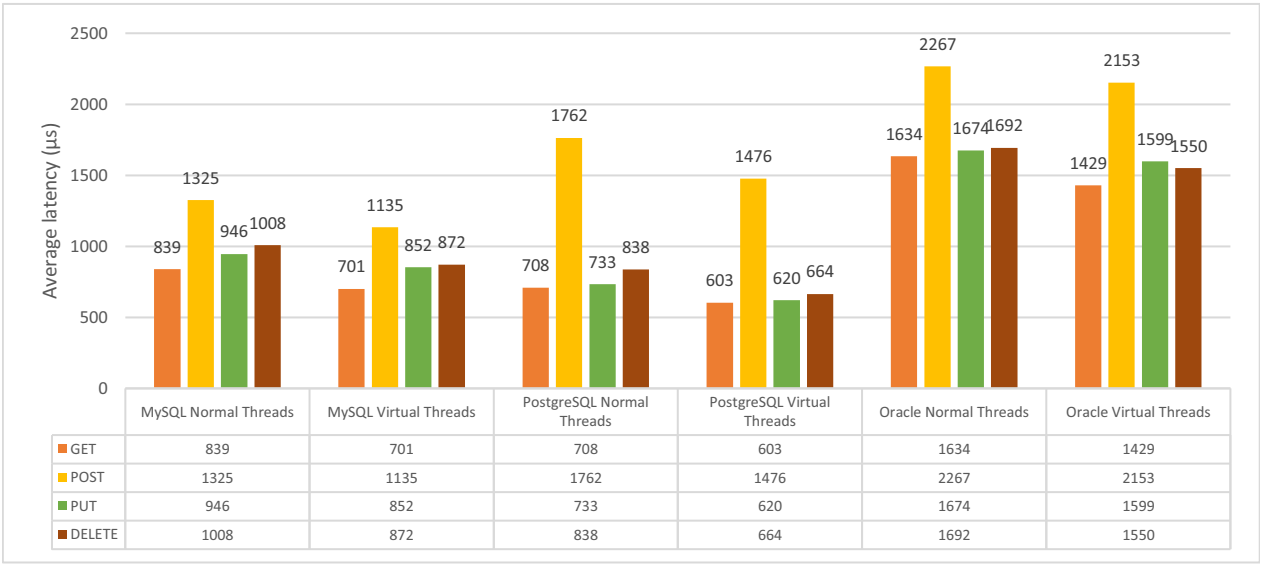
A. Results with application using relational databases

Performance outcomes were quantitatively assessed based on the response latency metrics for each request, measured in microseconds (µs), facilitating a comparative analysis of the efficiency and scalability of each database-driven system when interfaced with concurrent processing models. Figure 1 presents the visual comparison of average latency results on applications using relational databases juxtaposing their performance when interfaced with conventional threading mechanisms and Virtual Threads. The column size of each graph represents the latency metrics related to CRUD operations performed by each database-driven application.

Our preliminary findings indicate a noticeable enhancement in performance across test applications with relational databases when employing Virtual Threads for most CRUD operations. Application with PostgreSQL demonstrated the highest difference in efficiency, exhibiting the lowest latency times for GET, PUT, and DELETE operations under both threading paradigms. In contrast, application with MySQL presents the lowest latency times handling POST requests. Oracle database exhibits the slowest response times across all operations, probably given the relatively small dataset size (<14KB), which suggests a potential underutilization when using a smaller number of records in a database table.

In terms of performance details using Virtual Threads implementation, test application with PostgreSQL recorded the most significant latency reduction: a 14.83% decrease for GET requests, 16.23% for POST, 15.42% for PUT, and an impressive 20.76% for DELETE operations, marking the highest efficiency gain among the evaluated cases. Test application with MySQL also showcased notable improvements, notably a 16.45% latency reduction for GET requests, but it experienced the smallest gain in PUT request handling of 9.94%. Test application using Oracle observed the least performance increase, with a modest 4.49% enhancement in PUT operations and a 12.55% improvement in GET requests, underscoring its relatively muted response to Virtual Threads compared to other test applications with relational databases.

The preliminary performance results reveal that GET and DELETE operations benefited the most significantly from Virtual Threads improvements across all test applications with relational databases, with an average performance uptick of 14.61% and 14.27%, respectively. Conversely, PUT operation experienced the least average performance gain of 9.95%. Our initial findings of these experiments present heightened efficacy of test applications



	MySQL Normal Threads	MySQL Virtual Threads	PostgreSQL Normal Threads	PostgreSQL Virtual Threads	Oracle Normal Threads	Oracle Virtual Threads
GET	839	701	708	603	1634	1429
POST	1325	1135	1762	1476	2267	2153
PUT	946	852	733	620	1674	1599
DELETE	1008	872	838	664	1692	1550

Figure 1. Average latency of CRUD operations for test applications with relational databases (MySQL, PostgreSQL and Oracle)

using MySQL and PostgreSQL databases when dealing with smaller datasets and high request rates.

A. *Results with application using non-relational databases*

As presented in Figure 2, an enhancement in performance with the application of Virtual Threads was observed in all test cases based on non-relational databases. The performance of test applications with non-relational databases reveals that the test case with Cassandra achieves the highest efficiency across all CRUD operations with both threading models, followed by the test application with MongoDB, which slightly outperformed the test application with Neo4j in most operations.

Test application with Cassandra presented the most notable improvement, with average performance increase quantified at 24.6% for GET, 10.44% for POST, 15.29% for PUT, and 15.49% for DELETE operations, marking the most substantial gains across all database systems, particularly in GET operations. Test applications with MongoDB and Neo4j also displayed performance increments in GET operations at 16.45% and 12.55%, respectively, and the lowest increase was in PUT operations at 9.94% and 4.49%. Among non-relational test cases, application with Neo4j experienced the most modest overall enhancement in combination with Virtual Threads.

The assessment indicates that test applications using relational databases generally exhibit the highest performance increase in GET operations, with consistent gains across other operations. The tested scenarios focused on query operations; thus, database optimization and increasing the data load could yield divergent performance outcomes.

When considering the performance of relational and non-relational databases under higher request rate, PostgreSQL demonstrated the best performance for GET, PUT, and DELETE requests, and Cassandra had the best performance in serving POST requests. MySQL and PostgreSQL presented commendable results in GET, PUT, and DELETE operations but were outpaced by non-relational databases in handling POST requests, regardless of traditional threading or Virtual Threads implementation.

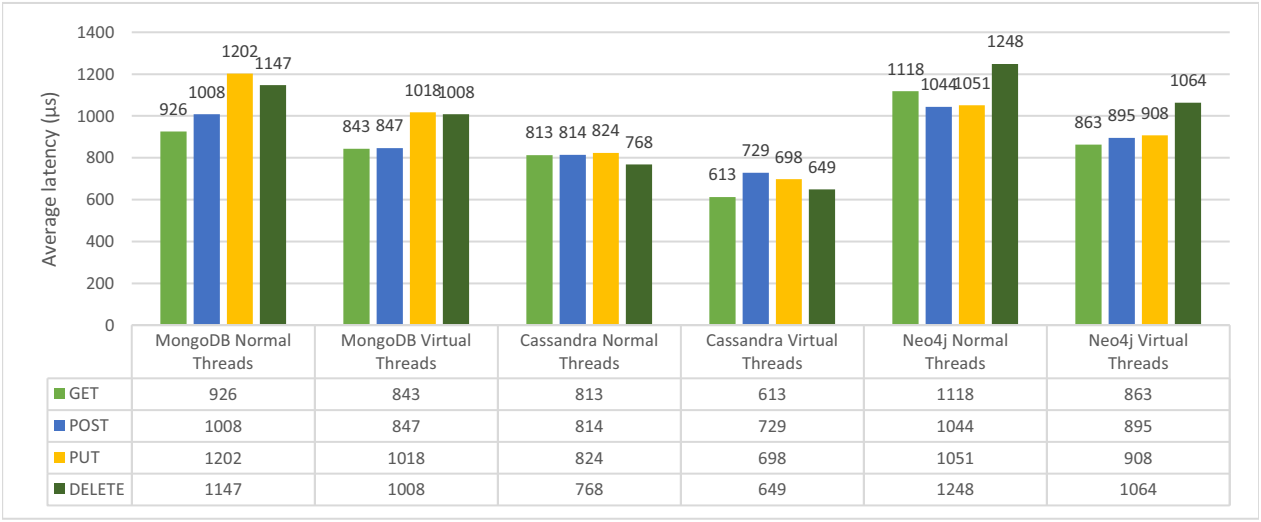
VI. Discussion and Future Work

Throughout the experiments, a consistent pattern of performance enhancement was observed with the introduction of Virtual Threads, underscoring their potential benefits. However, Virtual Threads exhibited a marginally reduced performance for some specific requests.

The preliminary results confirm a key rationale behind the development of Virtual Threads: to improve Java's capacity for managing I/O operations, particularly in the context of web applications. The data suggest a clear tendency for Virtual Threads to reduce latency in CRUD operations interfacing with a database.

When evaluating the performance of the test applications with selected relational databases, MySQL, PostgreSQL, and Oracle, the integration of Virtual Threads correlated with a reduction in latency, averaging approximately at 100 microseconds. The data indicate a particularly effective synergy between Virtual Threads and MySQL implementation, and PostgreSQL also demonstrated promising, with the exception of POST requests. However, to our surprise, the test application with Oracle did not show any substantial benefit from employing Virtual Threads, and this could require further exploration with larger datasets.

In the case of test applications with non-relational databases, Virtual Threads presented a more uniform pattern of performance, frequently outperforming or equating the efficiency of traditional threads. Test application with MongoDB showcased a comprehensive enhancement of Virtual Threads across CRUD operations, suggesting a more natural compatibility. Cassandra's test case reported a significant performance leap for GET requests, although an anomalous surge in latency for traditional threads suggests the possibility of an underlying anomaly in the test environment. In general, MongoDB has been recognized for efficiency in performing document search, storage capabilities, and intricate aggregation functionalities, while Cassandra demonstrated enhanced support sustaining its



	MongoDB Normal Threads	MongoDB Virtual Threads	Cassandra Normal Threads	Cassandra Virtual Threads	Neo4j Normal Threads	Neo4j Virtual Threads
GET	926	843	813	613	1118	863
POST	1008	847	814	729	1044	895
PUT	1202	1018	824	698	1051	908
DELETE	1147	1008	768	649	1248	1064

Figure 2. Average latency of CRUD operations for test applications with non-relational databases (MongoDB, Cassandra, and Neo4j)

performance scalability with the increment of threads, with architecture that supports multiple master nodes within a cluster configuration [13]. As a result, a transition to Virtual Threads could be advantageous for applications with MongoDB and Cassandra, potentially compensating for the latency disparities. Meanwhile, test application with Neo4j experienced a modest yet consistent improvement in CRUD operations when utilizing Virtual Threads.

Synthesizing observations from both applications using relational and non-relational database systems, Virtual Threads consistently demonstrate superior performance over traditional threads. The data also implies that Virtual Threads, with some adaptations, might be particularly well-suited for non-relational databases. This could be attributed to the more modern architecture of non-relational systems, which typically support a high number of concurrent connections. Additionally, the typical HTTP request size could also favor Virtual Threads, which are designed to operate efficiently with smaller memory footprints, aligning well with the constraints of HTTP data transmission.

The outcomes of the conducted experiments might be influenced by several extraneous factors beyond the utilization of traditional or Virtual Threads. All applications and databases were executed on a local test machine, so it may raise the question of Virtual Threads' efficiency over more expansive network infrastructures. Certain irregular latency peaks for some requests observed during testing could be attributable to extraneous background processes that impeded server performance. The interaction of JDBC drivers/connectors with Virtual Threads warrants scrutiny, as there is potential to restrict Virtual Thread efficiency by affixing to specific OS threads or through other means of latency induction. In the experimental setup, we limited the number of connections permissible for JDBC connector. Further analysis of the maximum number of platform threads and Virtual Threads manageable by a single JDBC connector for each database could provide a more comprehensive understanding of their respective capacities.

For the broader adoption of Virtual Threads in the enterprise markets, further testing is imperative. Future research should include an assessment of the maximum request size that Virtual Threads can efficiently process, as well as their impact on server longevity. Additionally, examining the long-term effects of Virtual Threads on memory usage and their implications for garbage collection and resource management will be crucial. It is also worthwhile to investigate the performance of Virtual Threads with different data types, such as the BSON format used by MongoDB, to discern how they can handle varying data structures emanating from distinct databases.

VII. Conclusion

The advent of Virtual Threads in the Java Virtual Machine heralds a significant step forward in structured concurrency, offering to elevate the performance of multi-threaded Java applications. This innovation promises to harness OS resources more efficiently, a leap facilitated by Project Loom and subsequent integration as a core feature in JDK 21. Our investigative foray into Virtual Threads has shed light on their potential to refine Java's concurrency operations within database-driven server applications, particularly those adhering to a thread-per-request model across both relational and non-relational databases. The empirical evidence amassed points to Virtual Threads' superior performance over traditional threads, particularly in high-throughput CRUD operations, albeit with nuances across different database systems and data volumes. These insights pave the way for broader application of Virtual Threads in contemporary server applications, marking a promising avenue for optimizing cloud-based services and enhancing resource management in complex computing environments.

References

[1] H. Daoud, and M. Dagenais, "Multilevel analysis of the Java Virtual Machine based on kernel and userspace traces," *Journal of Systems and Software*, vol. 167, 110589, 2020.

[2] P. Pufek, D. Beronić, B Mihaljević and A. Radovan, "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine," *43rd International Convention on Information Communication, and Electronic Technology (MIPRO)*, Opatija, Croatia, pp. 1539-1544, 2020.

[3] D. Beronić, P. Pufek, B. Mihaljević, and A. Radovan, "On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM," in *44th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, pp. 1939-1944, 2022.

[4] I. Vershinin and A. Mustafina, "Performance analysis of PostgreSQL, MySQL, Microsoft SQL Server Systems based on TPC-H tests," 2021 International Russian Automation Conference (RusAutoCon), 2021.

[5] C. Győrödi, D. Dumşe-Burescu, D. Zmaranda, R Győrödi, G., Gabor, and G. Pecherle, "Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application's Data Storage," Applied Sciences, vol. 10, no. 23, 8524, 2020.

[6] A.-M. Bonteanu, C. Tudose, and A. M. Anghel, "Multi-Platform Performance Analysis for CRUD Operations in Relational Databases from Java Programs using Spring Data JPA," *2023 13th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, pp. 1-6, 2023.

[7] A.-M. Bonteanu, and C. Tudose, "Multi-platform Performance Analysis for CRUD Operations in Relational Databases from Java Programs Using Hibernate," *In Lecture Notes in Computer Science*, pp. 275–288, 2023.

[8] M. Goeminne, and T. Mens, "Towards a survival analysis of database framework usage in Java projects," *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp 551-555, 2015.

[9] A. M. Sai, D. Reddy, P. Raghavendra, G. Yashwanth Kiran, and V. R. Rejeenth, "Producer-Consumer problem using Thread pool,". 3rd Int. Conf. for Emerging Technology (INCET), pp. 1-5, 2022.

[10] X. Qu, "Application of Java Technology in dynamic web database technology." Journal of Physics: Conference Series, Conf. Ser. 1744 042029, 2021.

[11] L. G. Wiseso, M. Imrona and A. Alamsyah, "Performance Analysis of Neo4j, MongoDB, and PostgreSQL on 2019 National Election Big Data Management Database," *6th Int Con. on Science in Information Technology (ICSITech)*, Palu, Indonesia, pp. 91-96, 2020.

[12] A. Gorbenko, A. Romanovsky, O. Tarasyuk, "Interplaying Cassandra NoSQL Consistency and Performance: A Benchmarking Approach," Bernardi, S., et al. *EDCC2020: Dependable Computing - EDCC 2020 Workshops*. Communications in Computer and Information Science, vol. 1279. Springer, 2020.

[13] J. M. Araujo, A. C. de Moura, S. L. da Silva, M. Holanda, E. Ribeiro, and G. L. da Silva, "Comparative Performance Analysis of NoSQL Cassandra and MongoDB Databases," 16th Iberian Conf. on Information Systems and Technologies (CISTI), 2021.