JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 35 (2025) 209–215
Received: 17 February 2025
Accepted: 25 March 2025

Comparative analysis of database access performance of the Hibernate framework and the Jooq library

Analiza porównawcza wydajności dostępu do bazy danych szkieletu programistycznego Hibernate i biblioteki jOOQ

Karol Hetman*, Marek Miłosz

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

Abstract
The subject of this paper was a comparative performance analysis of accessing relational databases in Java using two major Java libraries: Hibernate and jOOQ. For Hibernate, the main focus was on queries written with CriteriaBuilder. The study concentrated on key performance metrics, such as operation execution times and the number of operations processed per unit of time for DQL and DML operations. The measurements were conducted using the JMH library. The jOOQ library proved to be more efficient for simple and complex queries.

Keywords: Java; Hibernate; jOOQ; database access performance

Streszczenie
Tematem niniejszego artykułu była analiza porównawcza wydajności dostępu do relacyjnych bazy danych w języku Java przy użyciu szkieletu programistycznego Hibernate oraz biblioteki jOOQ. W przypadku Hibernate zdecydowano się na wykorzystanie głównie zapytań pisanych przy wykorzystaniu CriteriaBuilder. W badaniach skupiono się na kluczowych miarach wydajnościowych, takich jak czasy realizacji operacji oraz liczba operacji wykonywanych w jednostce czasu dla operacji typu DQL oraz DML. Do wykonania pomiarów wykorzystano narzędzie JMH. Lepszym narzędziem pod względem wydajności w przypadku prostych oraz złożonych zapytań okazała się biblioteka jOOQ.

Słowa kluczowe: Java; Hibernate; jOOQ; wydajność dostępu do bazy danych

*Corresponding author
Email address: karol.hetman@pollub.edu.pl (K. Hetman)

1. Introduction

With the increasing number of Internet users and the growing complexity of data driven applications, the importance of efficient use of this data and optimizing database access is becoming one of the key factors in the development of modern information systems. The efficiency of data processing has a significant impact on the quality of application, performance and end-user satisfaction. Web and mobile applications should provide low response times, which is especially important for them to remain competitive in the market.

In the Java ecosystem, which is the 3rd most popular programming language in the world according to the TIOBE index [1], many solutions have been developed to allow database communication. Those worth noting in particular are Hibernate and jOOQ (Java Object Oriented Querying). Both of these tools are mature products designed to interact with relational databases, but they differ significantly in their approach of operating on data. Hibernate is a typical ORM (Object Relational Mapping) tool, providing a high level of abstraction. It maps Java objects to relational database structures, offers out of box methods to perform basic database operations and automatic transaction management. This allows to reduce the amount of boilerplate code and increases the readability and efficiency of programmers. However, this convenience comes at the cost of greater overhead resulting from object relational mapping.

jOOQ library offers a different approach, which allows precise control over the generated SQL queries which can translate into better performance for applications with more complex database queries and high computational requirements.

2. Purpose, scope of work and hypotheses

The purpose of this study was to compare the aforementioned two different Java based database access technologies, Hibernate and jOOQ. The research aimed to determine which one offers a better performance for various scenarios, involving DQL (Data Query Language) and DML (Data Modification Language) operations. Furthermore, this paper also sought to identify the strengths and weaknesses of both of these technologies for different use cases.

The scope of work included preparing the test environment, selecting performance metrics such as response time and number of operations processed per unit of time and implementing benchmarks to allow accurate comparative tests. The tests were carried out using a publicly available database schema called Chinook database [2], that represents the structure of an online music shop. Various testing scenarios were done to replicate real world use cases for this kind of database.

As part of this paper, collected data was analyzed and visualized in order to facilitate their interpretation and the formulation of conclusions regarding the tested

technologies. This paper aims to confirm or refute below hypotheses:

H1. jOOQ offers lower average execution time compared to Hibernate.

H2. jOOQ provides higher throughput for different numbers of cores compared to Hibernate.

3. Literature review

The topic explored in this paper has been the subject of many other similar studies examining the impact of object-relational mapping on application performance.

B. Pllany [3] makes a comparison between Hibernate, EclipseLink and OpenJPA, examining the execution time of CRUD (Create, Read, Update, Delete) operations and the difference in performance with large data sets. His findings indicate that OpenJPA excels in DML operations, while Hibernate is better at SELECT type operations. Salahuddin Saddr and et al. [4] analyzed the performance of HQL (Hibernate Query Language) queries, pointing out the importance of properly managing EAGER and LAZY data loading configurations to optimize system performance. The results show that using the LAZY fetch technique can significantly improve application responsiveness by reducing unnecessary system load.

In her work, P. Węgrzynowicz [5] discusses design antipatterns in Hibernate that contribute to excessive query complexity and resource consumption. Her research highlights the importance of managing collection types in one-to-many joins, which can significantly reduce unnecessary operations in applications dealing with large datasets. An alternative to the popular ORM approach is to work directly with SQL, which is the subject of a thesis by Jound and Halimi [6]. The authors, using the Laravel programming framework as a basis, found that in large applications where performance is important, raw SQL queries have an advantage over ORM. Their study shows that raw SQL reduces the time needed for CRUD operations and is more efficient for handling a large number of complex database queries.

The analysis by Thampi and Ashwina [7] focuses on the performance of Hibernate and other ORM tools.Their study shows that Hibernate performs better in multi user environments due to its extensive caching mechanisms. However, the authors found that iBatis, a tool that allows writing queries that are similar to standard SQL, proved faster for simple operations due to lower overhead. P. van Zyl and colleagues [8] in their study compare Hibernate with the object-oriented database db4o. The results show that db4o outperforms Hibernate in operations like searching and modifying data. Hibernate, on the other hand, proved superior in simple queries due to the advantages of relational databases for straightforward operations. Colley, Stanier and Asaduzzaman [9] studied query performance of ORM tools. They pointed out that ORM simplifies application development by abstracting away database interaction, but this abstraction can introduce latency and reduce program performance in complex scenarios. Their results show the trade offs between development convenience and runtime efficiency while using ORM tools in demanding scenarios.

The article [10] focuses on the performance and flexibility of the jOOQ tool, showing its capability to provide full control over SQL queries. The authors point out that the library allows writing almost native SQL for database operations while facilitating their eventual optimization, if needed. The study presents examples of using jOOQ in various scenarios and compares its performance with ORM tools such as Hibernate. The findings highlight jOOQ's suitability for applications with high-performance requirements, where precise control over SQL execution and optimization is important.

The results of the cited articles provide insight into the trade-offs between ORM flexibility and the control offered by direct SQL management. They highlight the important aspects such as data loading strategy, caching mechanisms, and HQL query configurations, all of which impact performance.

Most of the existing research focuses on either a single tool or compares several ORMs with each other. In contrast, this paper addresses the topic of comparing the performance of an ORM (Hibernate) and jOOQ, which allows writing queries similar to raw SQL. In addition, for Hibernate, the CriteriaBuilder class was primarily used to create queries, ensuring that the queries written with both tools were as similar as possible. This allows, to largely ignore Hibernate's automatic query generation, which, as other articles on the subject already have shown, is often inefficient.

4. Tools

Prior to the study, a literature review was performed to analyze the tools used in this research. A description of their most important features is presented below.

4.1. Hibernate

Hibernate [11] is a comprehensive ORM tool that implements JPA (Java Persistence API) specification. It automates the mapping of POJO (Plain old Java objects) to tables of relational databases, and by this simplifying the software development process and minimizing the need to write complex SQL queries manually. It offers advanced features such as lazy loading, caching, and HQL, which allows developers to operate on data in object-oriented manner without directly using SQL.

4.2. jOOQ

The jOOQ tool [12] allows developers to have almost full control over queries, by enabling creation of queries that closely resemble native SQL, with the provision of mechanisms to verify their syntactic correctness. This increases the precision and efficiency of database operations. Due to these aspects, jOOQ is often chosen in applications with high performance requirements and where control over SQL queries is crucial.

5. Test environment

Both applications were run using OpenJDK 17.0.14, with initial heap size set to 256 MB and maximum heap size configured to 4 GB. Memory management relied on the default Garbage Collector (GC) provided by OpenJDK. Tests were run on Ubuntu 24.04.1 LTS x86_64 operating system with PostgreSQL running directly on bare metal to minimize latency and ensure optimal performance. To maintain consistency and reduce interference, only the essential processes required by the operating system were run during testing. For this study, two applications were developed using the Java programming language. These applications differ in the libraries used for database interaction and the implementation of the repository layer. In the application testing Hibernate, no dependencies specific for jOOQ were used, and vice versa. Both applications used common dependencies provided by Spring project: spring-boot-starter-web, spring-boot-starter-test, spring-jdbc, postgresql using their default versions provided by Spring. The specification of the test environment is available in Table 1, the versions of the software used in Table 2.

Table 1: Test environment hardware

CPU	Intel i5-4460 3.40Ghz
RAM	16GB (DIMM DDR3 1600MHz)
GPU	AMD ATI Radeon R9 380 4GB
Hard drive	WD Caviar Blue 1TB 3.5'' SATA III (WD10EZEX)

Table 2: Software used in research

Operating system	Ubuntu 24.04.01 LTS x86_64
PostgresSQL	16.4
Spring Boot	3.3.4
HikariCP	5.1.0
JMH	1.37
JMH Generator Annprocess	1.37
Hibernate Core	6.6.1.Final
jOOQ	3.19.13
jOOQ Codegen	3.19.13

The test data came from the publicly available Chinook database schema, which models a fictional music store with 11 tables. To ensure the representativeness of the tests, a script was implemented in the Golang programming language to generate additional records by permuting existing ones in the database. The final number of records for each table was: album - 50,000, artis - 580,000, customer - 950,000, employee - 100, genre - 25, invoice - 1,170,000, invoice_line - 2,100,000, media_type - 5, playlist - 5,000, playlist_track - 200,000, track - 2,100,000. Such amounts of data made it possible to reproduce a production-like environment and reduced the risk of fully loading the database into RAM. Figure 1 shows the schema of this database.

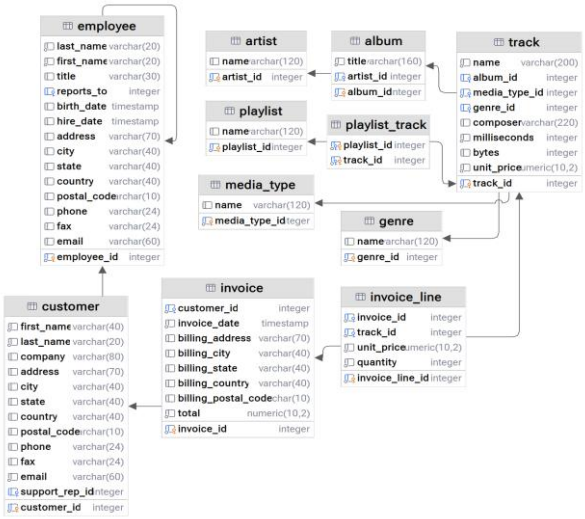

Figure 1: Chinook database entity schema.

6. Research methodology

The JMH (Java Microbenchmark Harness) library and JUnit framework were used to test the performance of Hibernate and jOOQ. These tools measured the average execution time of operations and throughput - the average number of operations per unit of time. JMH was configured for three warmup iterations and ten measurement iterations, each lasting 30 seconds. The warmup iterations were performed to prepare the JVM (Java Virtual Machine) and the code under test for accurate performance measurement, eliminating the impact of initial optimizations and other side effects. The warmup interaction allowed the JIT (Just-In-Time Compiler) to optimize key methods, load classes, initialize resources and stabilize memory allocations. During the measurement phase, JMH recorded actual values for execution times or throughput, saving results for each measurement iteration. Over the thirty second duration of a single iteration, JMH repeatedly executed the code fragment under test, recording the results for each time and then calculating the average. The tests were conducted in an environment with only the necessary services running to minimize the impact of system load. Results from all iterations were averaged to produce a single result for each test scenario.

For Hibernate, DML operations were performed using JPA specification compliant methods: *persist*, *merge* and *remove* for INSERT, UPDATE and DELETE operations, respectively. SELECT queries were implemented exclusively using the CriteriaBuilder class. This approach was chosen because, of all the available tools that the Hibernate framework has to offer, queries created using CriteriaBuilder are the closest to the

approach used in jOOQ. This ensured a more consistent and fair comparison of performance.

The tests included simple CRUD operations performed on individual tables: artist, customer, invoice, invoice_line and track. More complex SQL SELECT queries also operated on the genre, playlist and album tables. Tests with pagination and batch UPDATE, INSERT and DELETE operations were performed for data sets containing 100, 1,000 and 10,000 records, allowing performance to be evaluated under different workloads. Throughput was measured for simple queries, for the number of threads ranging from 1 to 4.

Complex scenarios consisted of:

1. Find total sales for every artist in the database.
2. Find all invoices with a higher total price than average.
3. Find all tracks with album and artist information.
4. Find customers by their first name, surname and email.
5. Find all tracks by selected artist.
6. Find tracks with album and artist information. Result will be paginated, with 50 tracks at one page.
7. Find invoice details for selected customers.
8. Find all tracks bought by selected customer.
9. Find all tracks from the playlist that are of the selected music genre.

7. Results

The results of the experiment are presented below. Figure 2 compares average execution for DML operations and figures 3 - 5 presents throughput for these operations, measured for number of threads ranging from 1 to 4. Figure 6 shows a chart of average execution time for SELECT operation, where the selected record was either random (Random ID) or always the same (Const ID), on Figure 7 is throughput comparison for SELECT with const ID. Figures 8 - 11 illustrate average execution time for batch operations: DELETE, INSERT, UPDATE and SELECT, respectively. Figures 12 - 14 represent comparison of average execution time for complex scenarios.

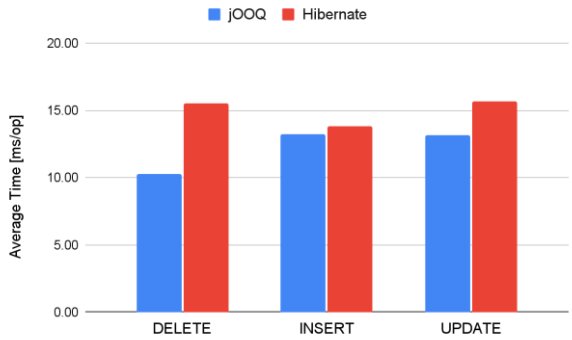

Figure 2: Average execution time for DML.

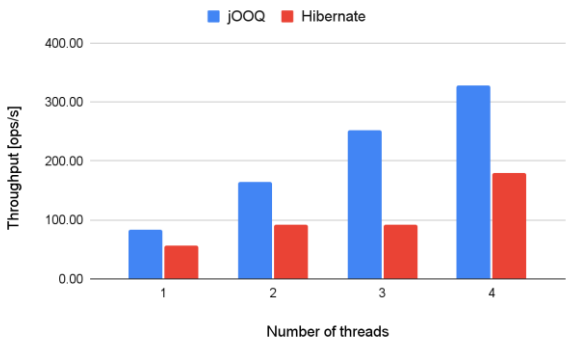

Figure 3: Average throughput for DELETE.

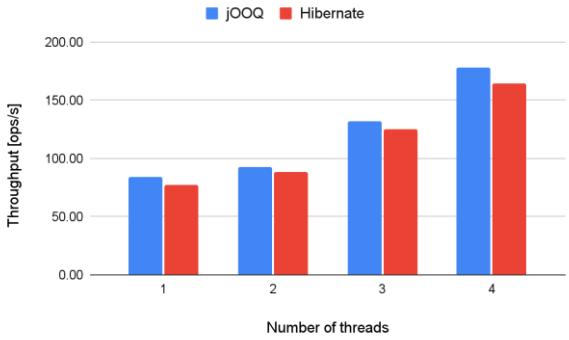

Figure 4: Average throughput for UPDATE.

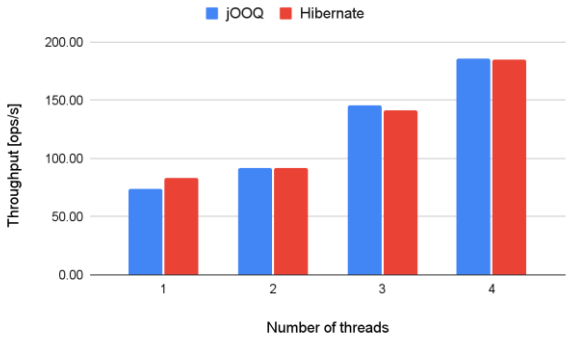

Figure 5: Average throughput for INSERT.

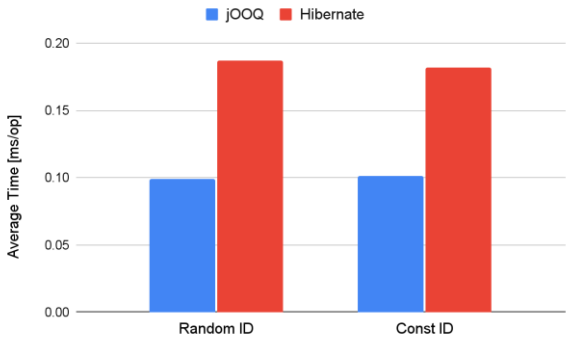

Figure 6: Average execution time for SELECT.

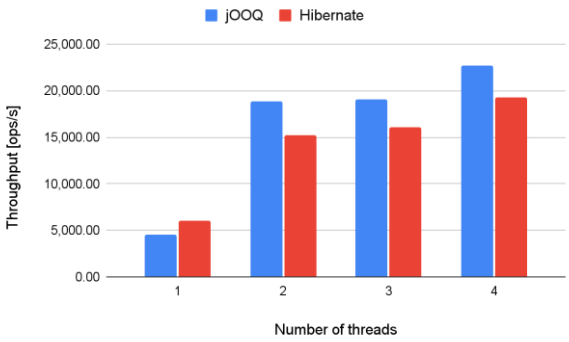

Figure 7: Average throughput for SELECT by const ID.

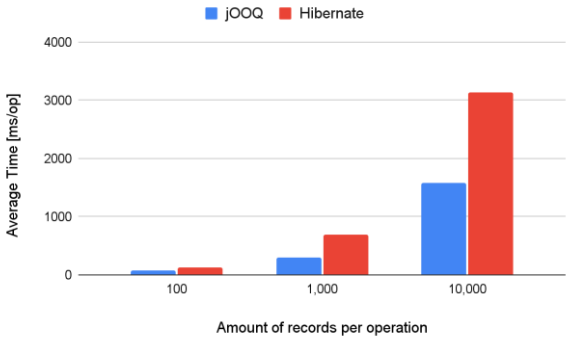

Figure 8: Average execution time for batch DELETE.

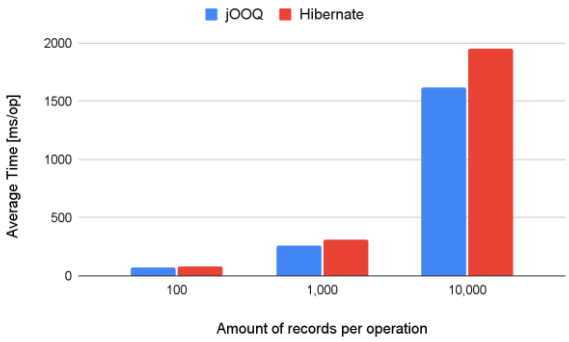

Figure 9: Average execution time for batch INSERT.

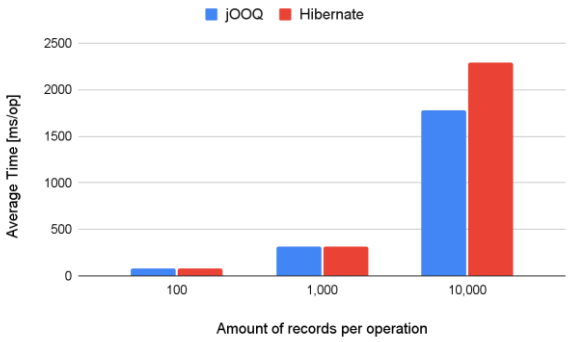

Figure 10: Average execution time for batch UPDATE.

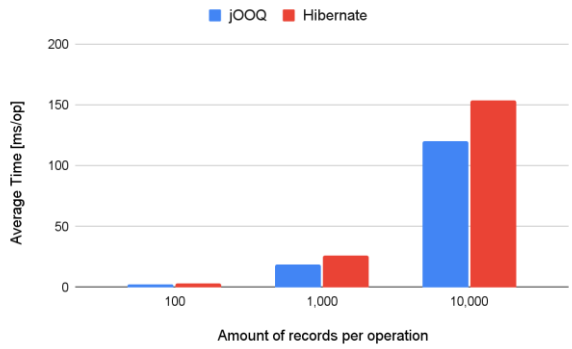

Figure 11: Average execution time for batch SELECT.

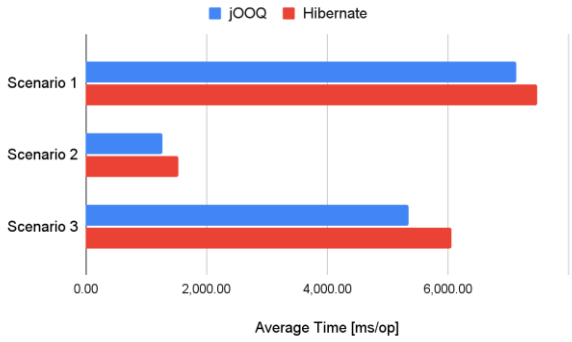

Figure 12: Average execution time for scenario 1, 2, 3.

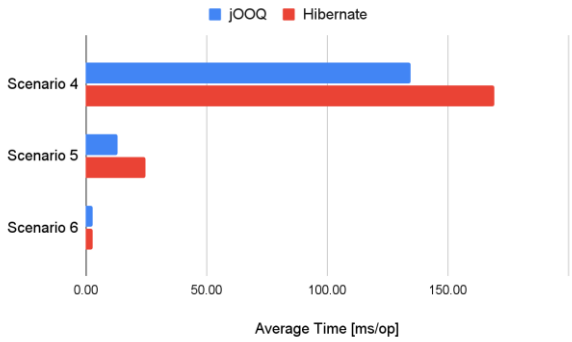

Figure 13: Average execution time for scenario 4, 5, 6.

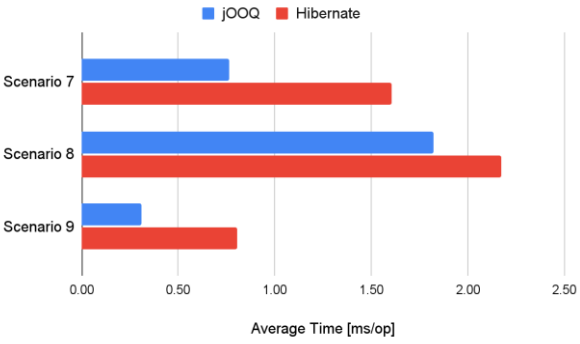

Figure 14: Average execution time for scenario 7, 8, 9.

8. Discussion

The conducted tests evaluated the performance of the Hibernate ORM framework and the jOOQ library by analyzing the average execution times and the average number of operations per unit time (throughput). The results clearly indicate performance differences in favor of the jOOQ library for all the types of operations studied - simple CRUD operations, batch CRUD operations and complex operations.

For simple operations, the results showed that jOOQ archives better average execution times for most of these operations, especially DELETE and SELECT. The average execution time for DELETE in jOOQ was 10.27 ms, which is visibly better than Hibernate (15.55 ms). The reason for this difference may be the lack of entity retrieval and mapping in jOOQ, which in Hibernate increases the operation time. A similar advantage of jOOQ was noticed in SELECT operations, where the average time was 0.1 ms (for a random ID), while Hibernate required as much as 0.19 ms. INSERT and UPDATE operations, although they are also more efficient in jOOQ, the difference is not as noticeable, because of the fact that for both libraries these operations are performed in a similar manner.

A throughput study for simple operations with number of threads ranging from 1 to 4, demonstrated jOOQ's advantage over Hibernate in the scalability area as well. For DELETE operation, jOOQ achieved higher throughput (approximately 300 ops/s) with 4 threads compared to Hibernate (less than 200 ops/s). A similar trend was observed for SELECT with a fixed ID, where jOOQ easily reaches 22,000 ops/s, while Hibernate reached about 19,000 ops/s. These results indicate that jOOQ makes more efficient use of multithreading and maintains higher throughput under concurrent workloads.

Batch operations performed for the number of records ranging from 100 to 10,000 indicated an advantage for jOOQ with a larger data set. For batch DELETE for 10,000 records required 1,573 ms, while Hibernate took significantly longer at 3,142 ms. For the batch INSERT, the differences were less visible for smaller data sets, but for 10,000 records jOOQ performed 17% better (1,619 ms vs. 1,956 ms). Batch UPDATE showed similar results for both tools with smaller data sets, but for larger numbers of records (e.g., 10,000) jOOQ clearly outperformed Hibernate.

In complex scenarios involving operations with multiple JOINs, aggregate functions (SUM, AVG, GROUP BY) and nested queries, jOOQ consistently performed better. Hibernate, using CriteraBuilder, generated more abstract queries, which introduced additional session and entity management overhead. For instance, in scenario 3, jOOQ completed the operation in approximately 4,000 ms, while Hibernate required more than 6,000 ms. The additional management mechanisms in Hibernate, while offering greater flexibility, negatively affect performance in more complex queries.

9. Conclusion

In this study we conducted a comparative performance analysis of the Hibernate and jOOQ libraries. Differences in their behavior were identified depending on the type of database operations. The results show that jOOQ, due to its closer integration with native SQL, performs better in simple operations and complex queries that require more advanced SQL features. On the other hand, Hibernate offers a higher level of abstraction that makes it easier to integrate these frameworks with applications and increases developer productivity, at the cost of some performance overhead.

Hypothesis H1 was confirmed, as the results for simple operations (especially DELETE and SELECT), batch operations and complex scenarios showed an advantage for jOOQ in terms of average execution times. The exceptions were some batch operations with small data sets, where the differences were marginal. Hypothesis H2 was also confirmed. For most operations, throughput in jOOQ increased proportionally to the number of threads, indicating better scalability.

The research showed that jOOQ is more suitable for applications requiring high performance, especially in multi-threaded environments and when processing large amounts of data. Hibernate, while offering more flexibility and convenience for developers, may be more useful in applications where performance is not a key factor. Future research should consider analyzing other metrics, such as memory and CPU usage.

Literature

[1] Programming language popularity index. TIOBE Index, https://www.tiobe.com/tiobe-index/, [01.11.2024].

[2] Documentation of database schema, Chinook, https://github.com/lerocha/chinook-database, [15.12.2024].

[3] B. Pllana, Performance Analysis of Java Persistence API Providers, In UBT International Conference (2018) 101, https://doi.org/10.33107/ubt-ic.2018.101.

[4] S. Saddar, J. Baloch, M. Sami, N. Pirzada, V. Khalique, A. Memon, Evaluating Performance of Hibernate ORM Based Applications Using HQL Query Optimization, Oriental journal of computer science and technology 11 (2018) 115-125, http://dx.doi.org/10.13005/ojcst11.02.07.

[5] P. Węgrzynowicz, Performance antipatterns of one to many association in hibernate, In Federated Conference on Computer Science and Information Systems (2013) 1475-1481, http://ieeexplore.ieee.org/document/6644211.

[6] I. Jound, H. Halimi, Comparison of performance between Raw SQL and Eloquent ORM in Laravel, Bachelor thesis, Blekinge Institute of Technology, Karlskrona, 2016, https://bth.diva-portal.org/smash/record.jsf?pid=diva2%3A1014983.

[7] S. M. Thampi, Ashwin A.K., Performance Comparison of Persistence Frameworks, arXiv, arXiv:2401.12345, (2007), https://doi.org/10.48550/arXiv.0710.1404.

[8] P. van Zyl, D. G. Kourie, A. Boake, Comparing the performance of object databases and ORM tools, Proceedings of the 2006 annual research conference of the

South African Institute of Computer Scientists and Information Technologists (2006) 1-11, https://doi.org/10.1145/1216262.1216263.

[9] D. Colley, C. Stanier, M. Asaduzzaman, The Impact of Object-Relational Mapping Frameworks on Relational Query Performance, In 2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE) (2018) 47-52, https://doi.org/10.1109/iCCECOME.2018.8659222.

[10] A. Sharma, P. N. Barwal, JOOQ - Java object oriented querying, International Journal of Research in Engineering and Technology 3 (2014) 315-319, https://doi.org/10.15623/IJRET.2014.0309049.

[11] Documentation of the programming framework, Hibernate, https://hibernate.org/orm/documentation/6.6/, [22.12.2024].

[12] Documentation of the library, jOOQ, https://www.jooq.org/doc/latest/manual/, [22.12.2024].