MicroStream vs. JPA: An Empirical Investigation

Benedikt Full, Johannes Manner[✉], Sebastian Böhm, and Guido Wirtz

Distributed Systems Group, University of Bamberg, Bamberg, Germany
`ben.fu@t-online.de`,
`{johannes.manner,sebastian.boehm,guido.wirtz}@uni-bamberg.de`

Abstract. MicroStream is a new in-memory data engine for Java applications. It directly stores the Java object graph in an optimized way, removing the burden of having to map data from the Java object model to the relational data model and vice versa, a problem well known as the impedance mismatch. Its vendor claims that their product outperforms JPA-based systems realized with Hibernate. They furthermore argue that it is well-suited for implementing microservices in a cloud-native way where each service complies with the decentralized data management principle of microservices.

Our work empirically assessed the performance of MicroStream by implementing two applications. The first one is a modified version of MicroStream's BookStore performance demo application. We used it to reproduce the data the MicroStream developers used as backing for their performance claims. The second application is an OLTP system based on the TPC-C benchmark specification.

MicroStream does not provide any sophisticated features for concurrent data access management. Therefore, we created two distinct MicroStream-based approaches for our OLTP application. For the first solution, we used a third-party transaction management system called JACIS. The second solution relies on structured modelling and Java 1.0 concurrency concepts.

Our results show that MicroStream is indeed up to 427 times faster when comparing the service execution time on the server with the fastest JPA transaction. From a user's perspective, where network overhead, scheduling etc. impact the overall server response time, MicroStream is still up to 47% faster than a comparable JPA-based solution. Furthermore, we implemented concurrent data access by using an approach based on structured modelling to handle lock granularity and deadlocks.

Keywords: Cloud-native applications · Java persistence · In-memory data engine · JPA · Concurrency control

1 Introduction

In 2019, the Java-native persistence solution *MicroStream (MS)* was released. It was integrated with Helidon, a set of open-source libraries for writing cloud-native microservices, in late 2021[1]. At its core, MS is a storage engine for managing and persisting Java object graphs. As it was developed specifically for handling Java objects, persisting data does not involve object-relational mapping (ORM). This fact is invoked by the framework developers as a major factor for MS's superior performance when compared to conventional relational persistence based on the Java Persistence API (JPA) standard. The developers of MS even claim that their persistence solution is "[...] up to 1000× faster than Hibernate + EHCache."[2] They support this by providing results acquired using their own, non-standardized performance evaluation solution, the *BookStore Performance Demo (BSPD)* application[3]. Our overall motivation for this work is to assess the marketing claim of MS as well as to compare the two persistence solutions with each other. We are aware that MS (in-memory) and JPA (ORM-based) solutions are two types of data management frameworks. Nevertheless both approaches allow a developer to work with their business objects in an object-oriented way. This is different from other in-memory data management solutions like Redis, where only key-value pairs can be stored, leading to a fragmentation of the domain model into disjunct objects. Furthermore the design principles of microservices, especially the decentralized data management principle, encourage developers to use the best data management solution for the use case at hand. This aspect fosters our motivation to look at MS as a candidate for a Java-native persistence solution.

To the best of our knowledge, no other publications have investigated this persistence solution and its vendor's claims regarding their product's performance. Therefore, the research questions of this work are:

– **RQ1** - Is a MicroStream-based solution up to a thousand times faster than a comparable JPA-based implementation utilizing Hibernate?
– **RQ2** - How can we achieve concurrency control for a mutable data model with the MicroStream in-memory data engine?
– **RQ3** - What are potential usage scenarios where MicroStream-based persistence should be used instead of JPA-based persistence?

Evaluating the performance of any component or system is rather challenging. There seems to be no general consensus on how performance data must be measured and interpreted [20]. Vendors sometimes provide custom applications which are supposed to highlight the strengths of their products, while at the same time ignoring or downplaying the products' weaknesses. For performance comparisons between their product and competing systems, vendors may use their own, non-standardized evaluation design implementations which raise questions regarding the bias and reliability of the data acquired. Furthermore, the performance of any system depends on the workload and application scenario [17].

[1] https://medium.com/helidon/helidon-2-4-0-released-18370c0ebc5e.
[2] https://microstream.one/.
[3] https://github.com/microstream-one/bookstore-demo-performance.

Benchmarks are tools used for evaluating and comparing the performance of similar systems. A benchmark should allow its users to measure performance in a standardized, reproducable, and simplified way [17]. The scope of a benchmark, and thus the applicability of its results, are usually limited to some specific usage scenario. Our research focuses on the context of Online Transaction Processing (OLTP) applications - software systems in which multiple clients can access resources concurrently.

For our work, we used a modified version of the BSPD application[4] to acquire some baseline performance data. We then implemented the *Wholesale Supplier (WSS)* benchmark[5], an OLTP benchmark based on the well-established, standardized TPC-C benchmark[6] [15]. This benchmark was then used to evaluate the performance of two different MS-based implementations in relation to a JPA-based implementation. Besides gathering and analyzing performance data, we share our expertise for identifying potential usage patterns and best practices for working with MS.

The paper is organized as follows. Section 2 describes previous work in the area of persistence solution evaluation and approaches to deal with concurrency control. Section 3 provides a more detailed introduction to the BSPD and WSS applications and how they were used to acquire performance data. This data is introduced in Sect. 4 and its implications are the foundation to answer our research questions in the subsequent part, Sect. 5. Besides answering the research questions, we also discuss potential threats to the validity of our work. Section 6 concludes the paper and provides an overview of possible future work.

2 Related Work

2.1 Performance Evaluation

Evaluating performance in the context of computer systems—and more specifically, persistence solutions—has been of concern to developers, vendors, and researchers for decades [17].

Benchmarks were developed to provide convenient means for evaluation and to enable fair comparisons of the performance of different solutions. Standardization efforts began during the 1970s [20], driven by groups and councils from industry and academia [17]. The Transaction Processing Performance Council (TPC) was formed in 1988 as a body for defining standards for evaluating the performance of systems in the context of OLTP applications. One of their most successful publications is the TPC-C benchmark, a specification-based benchmark for evaluating persistence solutions in the context of OLTP applications, released in 1992 [15].

[4] https://github.com/fullben/bookstore-demo-performance.

[5] https://github.com/fullben/java-persistence-benchmark.

[6] The specification for the TPC-C benchmark can be found at http://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp.

Besides standardized benchmarks published by councils such as the TPC, various research projects have released or used benchmarks. One of the earliest benchmarks looking into the performance of relational databases are the so-called *Wisconsin benchmarks* published in 1983 [2]. The *HyperModel benchmark* from 1990 was used to evaluate object-oriented database management system (DBMS) in the context of engineering applications [1]. Another important benchmark in this context is the *OO1* benchmark from 1992, which—like the previously mentioned HyperModel benchmark—can be used for evaluating persistence solutions in the context of engineering applications (e.g., CAD and CASE applications). Its authors—Cattell and Skeen—deemed all existing applications insufficient for evaluating database systems for this usage scenario and, therefore, developed their own benchmark [7]. Based on the OO1 benchmark, Carey, DeWitt, and Naughton developed *OO7*, another benchmark for evaluating the performance of object-oriented databases in the context of engineering applications, released in 1993 [5]. While OO7 was quickly adopted by various vendors of object-oriented databases, its authors hoped that they would be able to eventually pass on their benchmark to some standards body [6]. Although this has not happened to this day, besides vendors, various researchers have used the benchmark for their own research projects [9,10,16].

Besides performance-focused work, researchers have also published evaluations that primarily rely on the qualitative comparison of the features of the systems being evaluated [8,14]. Other works use both a benchmark-based performance evaluation and a feature comparison [4,16].

While most of the previously described works deal with the evaluation of persistence solutions, only a few have been performed in the context of the Java environment: Jordan used a set of criteria and a custom implementation of the OO7 benchmark to evaluate Java-based persistence technologies such as EJBs, JDBC, JOS, and JDOs [16]. Based on this work, Zyl et al. compared the performance of object-oriented databases and relational databases by using yet another, custom Java-based implementation of OO7 [24].

2.2 Concurrency Control

In database research, topics like the granularity of locks, transaction management, or principles such as ACID have been discussed in the context of concurrent data access management for decades [12]. In JPA-based solutions the concurrency handling of updating data is delegated to the DBMS. Modern in-memory databases have similar problems to solve [18]. Handling concurrency control in an optimistic way is often discussed based on a multiversion strategy [19].

Since MS does not expose any meaningful concurrency control features, users of the persistence solution are forced to rely on external transaction management systems with an adapter for MS, like the Java ACI Store (JACIS) library[7]. Alternatively, developers may take it upon themselves to implement thread-safe

[7] https://github.com/JanWiemer/jacis.

data access in their applications. For this, they can rely on Java language features such as locks and concurrent collections [11]. This leads to a system design where business logic and concurrency control concepts are mixed in the source code. Best practices and strict design rules are necessary to avoid concurrency errors which are hard to test and resolve at runtime.

3 Methodology

3.1 BookStore Performance Demo Application

The vendor of MS has published the so-called *BookStore Performance Demo* application on GitHub. This application is used to back their claims regarding the superior performance of MS when compared to JPA-based persistence on their website, see **RQ1**.

The application is implemented in Java 8 using SpringBoot and both MS and JPA for persistence. The JPA-based, relational persistence uses Hibernate as JPA implementation and a PostgreSQL DBMS for managing the relational database. The business model of the BSPD application is that of a company selling books in stores located in multiple countries. It is worth mentioning that the model structures for the MS-based implementation are largely immutable to increase thread-safety and ease the burden of manual synchronization.

At BSPD application startup, an initial set of model data is generated for the MS-based persistence implementation. Once written to storage, this data is then also written to the JPA-based implementation, thus ensuring that both persistence implementation variants have the same initial set of data. After this setup has been completed, users can use the Vaadin-based web interface of the application to trigger one of seven predefined read-only queries. The selected query is executed for both, the MS-based and the JPA-based persistence implementations, and usually repeated multiple times. The execution durations for these queries are then reported back and visualized in the web interface. The actual result data of the queries is ignored. And although the queries are designed to be parameterized, the application selects the actual parameter values to be used automatically.

We developed an extension of this application[8]. It makes no significant modifications to the behavior of the existing application components. For executing the seven defined queries parameterized, we added a dedicated service layer. This service layer allows for query execution against both, the MicroStream-based and the JPA-based data. We made these services available as part of a new API. The endpoints of the API can be used to trigger the queries with appropriate parameters, provided via HTTP request properties. Additionally, we wrote a JMeter script that can be used to simulate multiple clients interacting with this API concurrently. The clients use the API in a two-step process:

1. Setup phase: A set of data is acquired from the API in order to define the value ranges for the parameters of the queries.

[8] https://github.com/fullben/bookstore-demo-performance.

2. Measurement phase: Each client randomly selects one of the seven queries and randomly chooses valid parameters before calling the appropriate API endpoint.

With this performance measurement approach, more data can be generated than with the original implementation. This should potentially reduce the impact of errors introduced by sources of uncertainty such as the host platform or the JVM JIT-compiler activity during the initial moments of the application runtime [3].

3.2 Why Another Custom Benchmark?

As indicated in Sect. 2.1, there is a variety of benchmarks for evaluating persistence solutions. So why did we see the need for implementing our own, custom benchmark?

Solely relying on the BSPD application would not have been appropriate, as it is a non-standardized, vendor-provided solution.

Most of the benchmarks described in Sect. 2.1 focus on the area of engineering applications. As our goal was to use a benchmark relevant for OLTP applications, using benchmarks developed for evaluating the performance of persistence solutions in the context of CAD or related software was not an option. Besides this obvious mismatch in focus, OO7 and its predecessors were initially published during the early 1990s. As the field of computing is vast and evolves quickly, benchmarks must either evolve to remain relevant or risk becoming outdated [15].

We, therefore, decided to implement a custom benchmark modelled after the specification-based TPC-C benchmark. The business model and workloads of TPC-C defined by the specification are relevant for a typical OLTP use case. Additionally, the business scenario of the TPC-C benchmark requires a mutable data model, as opposed to the immutable data model of the BSPD application. Furthermore, as the benchmark is specification-based, users must create a complete implementation themselves, allowing for a high degree of freedom in regard to technologies used by the benchmark implementation.

It has to be mentioned that the WSS benchmark is not fully compliant with the TPC-C benchmark specification. The reasons for this can primarily be found in our disagreement with certain requirements and structures defined in the specification. The specification heavily relies on the terminology of the relational data model. For example, it defines many primary composite keys for the data model entities. While this approach may have appeared intuitive in 1992, we were able to convert it to an object-oriented model. This allowed us to drop the foreign keys since these keys represent other objects which are class members in our approach. We also modified the overall data model by removing a model object we deemed unnecessary (*NewOrder*, used to explicitly indicate that an order is new and for artificially providing an opportunity for deleting data) and adding two new objects (*Employee* and *Carrier*). These two map entities which are implicitly part of the TPC-C business model, but not modelled as entities in the benchmark specification.

3.3 Wholesale Supplier Benchmark

Just like the TPC-C specification, the WSS benchmark models the order-entry system of a wholesale supplier.

In the business model of our WSS application, the employees of a company use computer terminals to perform their work tasks, such as adding a new order of a customer or updating an order's payment data. These tasks are referred to as *transactions*.

Table 1. The business transactions of the WSS benchmark.

	Transaction type	Read-only	Minimum % of mix
WSS1	Order-Status	Yes	4
WSS2	Stock-Level	Yes	4
WSS3	New-Order	No	45
WSS4	Payment	No	43
WSS5	Delivery	No	4

The terminals are clients of the main application that implements the business logic and manages data maintained by some persistence solution. For communication with the terminals, the application exposes a web API, secured with basic authentication. The API has two distinct sections: The first provides a set of read-only endpoints for accessing most of the data maintained by the application. The second section has endpoints that enable the parameterized execution of five predefined business transactions which are listed in Table 1, together with their execution probability. For referring to these transactions in later sections of the paper, we numbered them with our application prefix (WSS1 to WSS5). Of these five transactions, two are read-only and three are read-write actions. The server is implemented in Java 11 using SpringBoot. We implemented the application by providing two generic core modules, on which actual WSS server implementations must be based. The first of these two is a component for data generation which can be used to create the initial population of the database in a persistence solution independent model representation structure. This component relies on the JavaFaker library[9] for some of the random data generation. This data can then be converted to any solution-specific model. In the second component, we defined the overall architecture of the server. This includes the API structure, security, data transfer structures, and services.

For the WSS benchmark, we created three actual implementations of the WSS server:

1. *JPA*: Uses JPA-based persistence, with Hibernate as JPA implementation. Spring Data JPA is used for data access. The relational database is managed by a PostgreSQL DBMS. Concurrent data access is facilitated by employing the transaction mechanism defined by JPA.

[9] https://github.com/DiUS/java-faker.

2. *MS-JACIS*: Relies on MS for data storage and uses the JACIS library for data access synchronization by means of transactions on transient data. As JACIS uses Java object cloning for transaction isolation, we were forced to completely decouple the data model classes of this implementation. In any regular implementation (e.g. JPA-based implementation) an *Order* class would have a field referencing the appropriate *Customer* object. But in the case of this implementation, the *Order* only has a field containing an artificial identifier for the related *Customer* object. This approach makes simple object graph navigation impossible, which has significant performance implications.
3. *MS-Sync*: Also uses MS for data storage. Concurrent data access is achieved by using synchronization features provided by the Java environment. Primarily, locks and the *synchronized* keyword are used with the aid of Fig. 1.

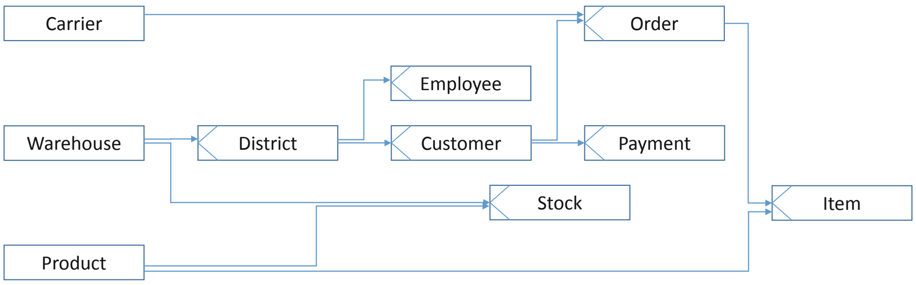

Fig. 1. Simplified Structured Entity Relationship Model of our WSS application.

For the MS-Sync variant, we analyzed the data model by using the Structured Entity Relationship Model (SERM) notation format [23], as depicted in Fig. 1. In this diagram, we have independent entity types like the carrier or the warehouse, which can also be identified by the shape of their boxes. Furthermore, there are entity-relationship types such as the district, which is dependent on the warehouse and would therefore hold the foreign key of the warehouse in a relational model. This notation gave us a direction of dependence which was helpful when determining the ordering of our locks in the concurrent Java implementation of our application. It is important to note that we used a simplified version of SERM. The arrows in Fig. 1 do not indicate the cardinality since we only want to visualize the interdependence of the individual classes of the data model.

Besides the server, we developed a JMeter script that can be used to simulate the employee terminals. Just as in the case of the BSPD application framework, each simulated terminal has two main phases of execution: the setup phase and the measurement phase.

For each of the actual server implementations, we have also provided a Docker Compose file which can be used to configure and launch the server and any necessary auxiliary systems as Docker containers.

3.4 Experimental Setup

For our experiments we used two bare-metal Linux machines with an Ubuntu 20.04 server image. The primary machine (*H90*) was a Fujitsu Esprimo P757 with an Intel Core i7-7700 CPU with 4 cores and 210 GFLOPS peak performance. We used a LINPACK benchmark to assess the peak performance and to verify the linear scaling behavior of our machines [22]. H90 had 32 GB of RAM and used a SSD with 256 GB as primary drive. The other machine, referred to as *H50*, was a Fujitsu Esprimo P700 with an Intel Core i7-2600 CPU with 4 cores and approximately 92 GFLOPS peak performance. It had 16 GB of RAM and a 240 GB SSD as primary drive.

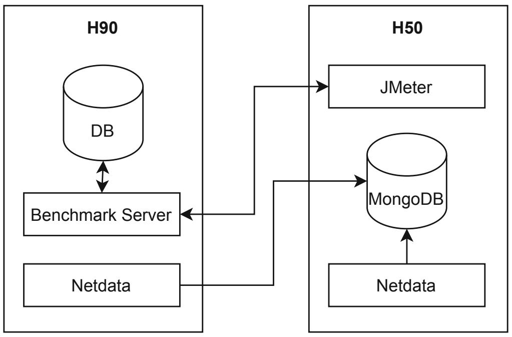

Fig. 2. Overview of the experimental setup, consisting of two physical machines. Note that the *DB* on H90 was, depending on the actual setup, either a SQL-based DBMS or the files (database) used by MicroStream to store data.

For monitoring, Netdata[10] was installed on both machines. Both Netdata agents sent their recorded data to the MongoDB instance on H50 once per second[11].

We used version 1.1.1 of the BSPD[12] and version 2.1.1 for the WSS application[13]. Both the BSPD and WSS benchmark are similar in their overall structure. They both have a Java application managing data operations and a JMeter script simulating clients interacting with this application. Due to this, the setup for measuring the performance with the two systems was very similar. We used the

[10] https://www.netdata.cloud/.

[11] Netdata claims that it only consumes 1% CPU utilization of a single core (https://github.com/netdata/netdata).

[12] https://github.com/fullben/bookstore-demo-performance/releases/tag/1.1.1.

[13] https://github.com/fullben/java-persistence-benchmark/releases/tag/2.1.1.

medium data generation option for BSPD. For the WSS, we scaled our model by changing the warehouse count, as defined by the TPC-C specification. Overall, we generated over 2.5 million objects: 5 warehouses, 50 districts (10 per warehouse), 50 employees (one per district), 100,000 products, 150,000 customers, and 150,000 orders. The remaining objects were order items, stock information, and payments. The impact of these settings on the used memory for the different applications will be discussed later.

Since we wanted an isolated workbench for the benchmark servers, we only deployed the benchmark server (BSPD or WSS) and their respective database on H90. The JMeter instance for executing the appropriate client-simulating script was installed on H50 and invoked the queries via the previously mentioned server APIs. This setup is depicted in Fig. 2.

Our measurement methodology focused on two metrics. Firstly, we recorded the user-perceived server response time via JMeter. Since this User-perceived Response Time (URT) contains a lot of uncontrollable effects like the physical transmission time, the middleware layers of our application etc., we also decided to additionally wrap the method call to the service method within the business logic layer to measure the Server Processing Time (SPT). This processing time value only included the actual time the business logic took to process the request. We used JMeter to save these two metrics and other data to a CSV file. For both applications, we simulated concurrent users executing the queries.

In the case of the BSPD application, we performed two distinct types of executions: one targeting the data persisted using MS, and another one aimed at the data maintained by the JPA-based persistence implementation. Each of these runs were executed twice to ensure that the data remained consistent. Both data sets proofed to be very similar, thereby indicating reproducibility of our results. We therefore used only the data from one of the runs for the evaluation included in this paper. For the WSS benchmark, we performed three distinct types of runs, one for each of the three implementations: *JPA*, *MS-JACIS*, and *MS-Sync*. As with the BSPD runs, we also performed each of these runs twice to ensure data consistency. After each run, we shut down the containers on the H90 machine and deleted the volumes containing the data written by the persistence solution of the current application implementation.

4 Results

All collected data and some diagrams visualizing CPU utilization, memory, disk IO, bubble plots for the different runs and applications as well as the scripts we used for generating the tables and plots can be found on our raw data page[14], where you can also download all data. For the discussion in this paper, we only used a subset of this data. CPU utilization, memory, and disk IO were measured for the machines in total since there are no other applications running on the machines apart from JMeter on H50 and the benchmark server on H90 as depicted in Fig. 2.

[14] https://spboehm.github.io/jpa-microstream-doc/.

In the BSPD application, the CPU utilization when using the JPA-based solution (∼20%) was quite different from that of the MS-based implementation (∼8%). This additional CPU usage in the case of the JPA-based solution is most likely caused by the DBMS and ORM overhead. In both cases, approximately 3,600 MB of RAM were occupied. Our WSS applications had a low CPU utilization (in all cases <5%), but varying memory demands. The JPA-based solution consumed the least amount of memory with ∼5,725 MB, whereas the MS-JACIS implementation consumed ∼11,600 MB of RAM. The MS-Sync solution used ∼9,500 MB. Comparing this last value to those of the other solutions, we see that the in-memory data engine requires much more RAM than the relational database. Furthermore, the memory overhead of decoupled data model in the JACIS variant becomes evident.

Table 2. BSPD performance data for JPA and Microstream. We used our server processing time (SPT) metric to measure the execution time.

JPA/MS	Median (values in millisecond)	Speed-up
[**BSPD1**] (6931/8380)	68.12/2.84	24.03
[**BSPD2**] (6935/8383)	3.64/0.91	3.99
[**BSPD3**] (6934/8382)	7.87/0.93	8.42
[**BSPD4**] (6936/8385)	2.8/0.12	23.3
[**BSPD5**] (6931/8376)	38.24/14.59	2.62
[**BSPD6**] (6929/8376)	305.06/0.72	426.61
[**BSPD7**] (6933/8381)	3.26/1.11	2.93

Table 2 summarizes the measured query processing times from our BSPD application. Each line of the table includes abbreviations representing the seven queries, *get book sales* (BSPD1), *get books by title* (BSPD2), *get books in price range* (BSPD3), *get customer page* (BSPD4), *get employee of the year* (BSPD5), *get purchases of foreigners* (BSPD6), and *get revenue of a shop* (BSPD7). In parentheses after the transaction identifier, the number of requests made per solution is depicted (JPA value first, followed by the corresponding MS value). The execution time of JPA requests is higher than that of MS requests, which explains the different number of requests as we used a fixed experiment duration. The last column shows the speed-up of our MS-based solution compared to the JPA-based solution for the BSPD application. We submitted the requests for every user in sequence. So one user of our application does only make a single request at a time. To stress the concurrency aspect, we configured JMeter with ten concurrent users.

We used R for data evaluation and to generate boxplots to visualize our measurements. Computing only the arithmetic mean for our transactions was too coarse-grained and over-represented outliers. Therefore, we decided to include the median for the BSPD application as shown in Table 2.

Table 3. Raw data of the boxplots from Fig. 3. The transactions are as follows: WSS1-GET order-status, WSS2-GET stock-level, WSS3-POST new-order, WSS4-POST payment and WSS5-PUT delivery. After the transaction identifier, the second line in the table header are the number of transactions executed for JPA, MS-JACIS and MS-Synch during our eight hours experiment. The first line of each cell contains the server-side processing time (SPT) in milliseconds for the individual solutions. Line two represents the slowdown (red) and speedup (green) of MS-JACIS and MS-Sync compared to JPA. Lines three and four follow the same structure as one and two but are based on the response times measured client-side (URT).

JPA/MS-JACIS/MS-Sync	[WSS1]	[WSS2]	[WSS3]	[WSS4]	[WSS5]
	(479/479/479)	(478/478/479)	(5388/5386/5390)	(5147/5145/5148)	(479/478/479)
Lower whisker	7.22/76.3/0.01 10.56/580.19 81/145/66 1.79/1.23	31.61/134.63/1.77 4.26/17.86 107/205/70 1.92/1.53	13.76/29.19/3.62 2.12/3.8 88/97/72 1.1/1.22	8.13/5.41/3.32 1.5/2.44 83/74/73 1.12/1.14	52.02/138.93/20.12 2.67/2.59 126/208/89 1.65/1.42
Lower quartile	8.15/85.06/0.03 10.44/307.81 85/157/73 1.85/1.16	35.55/144.84/1.96 4.07/18.17 113/218/76 1.93/1.49	18.96/33.72/4.58 1.78/4.14 97/106/78 1.09/1.24	9.74/8.17/4.76 1.19/2.05 87/81/79 1.07/1.1	58.26/148.97/21.09 2.56/2.76 135/221/95 1.64/1.42
Median	8.65/87.8/0.03 10.15/264.3 87/162/76 1.86/1.14	36.77/148.25/2.03 4.03/18.14 115/223/78 1.94/1.47	21.72/35.16/4.94 1.62/4.39 100/109/80 1.09/1.25	10.41/9.96/5.22 1.05/1.99 89/84/81 1.06/1.1	60.44/152.12/21.52 2.52/2.81 138/226/97 1.64/1.42
Upper quartile	9.27/91.09/2.63 9.82/3.53 88/165/78 1.88/1.13	38.19/152.05/2.09 3.98/18.25 117/227/80 1.94/1.46	24.25/36.75/5.24 1.52/4.63 103/112/82 1.09/1.26	11.03/11.69/7.42 1.06/1.49 90/86/83 1.05/1.08	62.58/155.68/22.08 2.49/2.83 141/230/99 1.63/1.42
Upper whisker	10.69/99.45/6.11 9.3/1.75 92/176/85 1.91/1.08	42.15/162.32/2.27 3.85/18.58 123/239/86 1.94/1.43	32.14/41.29/6.21 1.28/5.18 112/121/88 1.08/1.27	12.87/16.92/11.3 1.31/1.14 94/93/89 1.01/1.06	68.72/165.09/23.55 2.4/2.92 150/243/104 1.62/1.44

For WSS, we included all boxplot details for the quartiles (25%, median, 75%) and the whiskers (max. 1.5 times the size of the box).

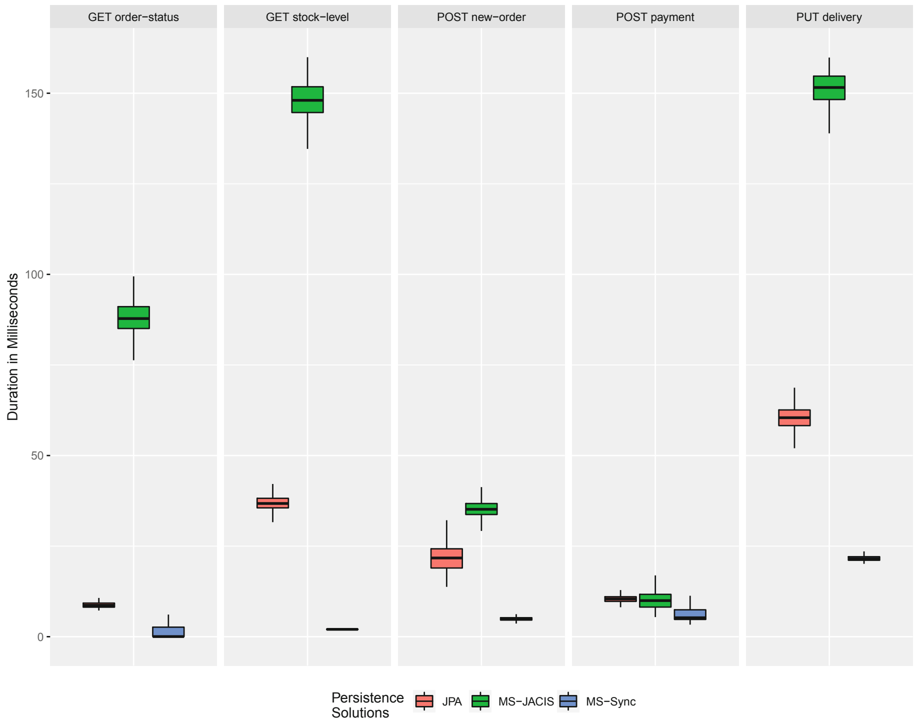

Fig. 3. Wholesale Supplier performance data of the five transactions depicted as boxplots for JPA, MS-JACIS and MS-Sync. We used our server processing time (SPT) metric to measure the execution time.

Figure 3 depicts the results of our WSS application benchmark, while Table 3 shows the raw boxplot data. We see for all transactions that our synchronous implementation with basic Java concurrency features is the fastest compared to the JPA and MS-JACIS implementations. Furthermore, MS-JACIS performed worse for most of the transactions, despite WSS4, leading to a consistent winner's podium for most transactions. Another view on the same data is presented in Fig. 4, where we see the server execution time over time when benchmarking our WSS application. For a better resolution of the Figure, we decided to exclude 0.2% of the outliers. The execution times for the JPA-based solution decreases slightly at the beginning when the JIT compiler still optimizes code and stabilizes after two hours. For the in-memory solution, only a minor increase is visible.

The structure of Table 3 is the same as for Table 2. WSS1 to WSS5 are in the same order as the headlines of the boxplots in Fig. 3. Each cell consists of four lines of data. The first line contains the processing time on the server (SPT) for

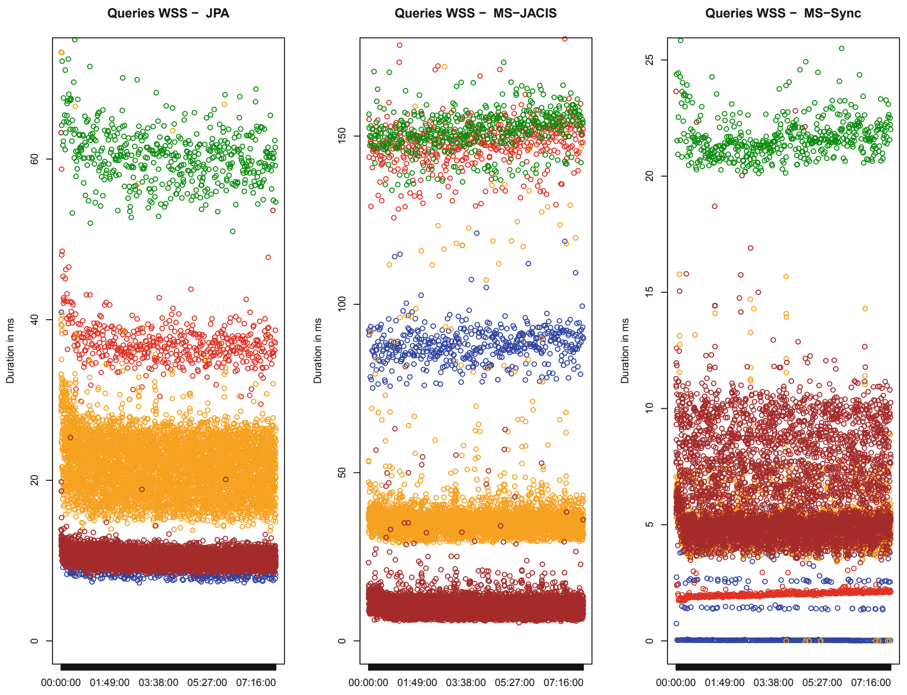

Fig. 4. Wholesale Supplier business transactions: *Order-Status* (blue), *Stock-Level* (red), *New-Order* (orange), *Delivery* (green), and *Payment* (brown). (Color figure online)

JPA, MS-JACIS, and MS-Sync requests. The second line compares MS-JACIS and MS-Sync to JPA. Green values indicate that the corresponding solution is faster by a factor of x compared to JPA, whereas red values stipulate that the solution is slower by a factor of x. The next two lines of each cell show the client-side measured response times (URT). This user-perceived performance includes network transfer, scheduling within the application, etc.

5 Discussion

5.1 MicroStream vs. JPA

First, we want to address MS's claim to be a thousand times faster than a Hibernate-based solution. Table 2 shows the adapted BSPD results. We can see that transaction **BSPD6** experienced the most significant speedup. Using the median values, MS is over 400 times faster than the JPA solution. This query navigates many nested objects which need to be read from the relational database via complex joins, whereas the MS solution can work on the Java object graph by using the Java Streams API. For all other queries executed by the BSPD, MS is faster than the JPA-based solution, but only by factors of tens, not thousands. This insight can be used to partially address **RQ1**. In order to provide a complete answer to the research question, it is important to consider another aspect. In the preceding parts of the discussion we referred to the processing time on the server. For a realistic scenario, we argue that the user-perceived performance must be compared. We did not use the user-perceived response time for a BSPD comparison since the response time measured by JMeter is only recorded on a millisecond basis with integer precision. This distorts the comparison with the server processing time which is sometimes only a small fraction of a millisecond like in **BSPD4**. We also looked at the user-perceived performance (UPT), which on average is a few milliseconds higher than the values measured server-side. The response handling and scheduling on the server adds about 5 ms per query (median of all queries).

Therefore, to fully address the first research question, in addition to the data acquired with the BSPD application, we must also consider the results gathered with the WSS application which contains a mutable data model. Additionally, as mentioned in the concurrency control Sect. 2.2, MS does not offer any sophisticated concurrency control or transaction management facilities. For this reason, we decided to use a suitable transaction framework with a MS adapter (implemented in the MS-JACIS variant) as well as a solution based on low-level synchronization utilizing the Java 1.0 capabilities (implemented in the MS-Sync variant). Especially for transaction WSS1, we see a situation where MS performs best, see boxplots in Fig. 3 and detailed data in Table 3, when looking at the first line of data in each cell which represents the service time on the server (SRT). This is similar to the BSPD application, where MS is a few hundred times faster than the JPA-based implementation. On the other side, MS-JACIS performs worse by a factor of 8–11 compared to JPA, and even worse when comparing it to MS-Sync. JACIS appears to be currently the only available solution for using transactions on transient objects in the context of MS-managed data. The performance data we acquired indicates that JACIS as a third-party transaction middleware cannot compete with JPA-based solutions. Therefore, we exclude the JACIS-based solution (MS-JACIS) and corresponding data from all further analysis.

When looking at user-perceived performance in the third and fourth line in each cell, the quotient is not greater than 1.47 (median of **WSS2**) for MS-Sync compared to JPA. Also, when looking at the millisecond values, it is evident that the response overhead ranges between 65 and 85 ms and has a dominant impact on calculating the quotients and the speedup for the user. Nevertheless, based on our results, we have to conclude that MS is not 1000$\times$ faster than a JPA solution. This gives an answer to **RQ1**. We found only a few transactions (**BSPD6** and **WSS1**) where MS is a few hundred times faster when assessing the service execution time and none where it is faster by a factor of a thousand. Furthermore, it must be considered that these speedups are not the actual, user-perceived times. In the case of user-perceived response times, we see an improvement between 10% (median of WSS4) and 47% (median of WSS2) when comparing JPA and MS-Sync. Therefore, MS appears to be capable of outperforming JPA-based persistence, albeit not by as much of a margin as claimed by the vendor of MS.

In a first version of this paper we experienced a linear increase in execution time for WSS5 - the delivery transaction. The first executions took ~75 ms and after 6 h benchmark, the execution time increase linearly to ~130 ms. At the beginning the assumption was that the increase is caused by WSS3, the new-order transaction, where over time the number of orders increased and therefore the filtering and sorting is more time consuming. Considering the number of initial orders (150,000) with the newly created orders (3,376), the increase was not justifiable. A detailed description and figures for this step-by-step investigation can be found on our GitHub IO Page[15]. When searching for the cause after looking at database fragmentation, index fragmentation and the LAZY and EAGER loading capabilities of JPA, we changed the service implementation as well as the native JPA query. Our assumption was that the many database queries and the ordering within one query (ORDER BY SQL feature) caused the performance problem. Connecting to a remote machines causes IO waits, therefore we reduced the number of database queries to a minimum and executed the benchmark again. The collected performance data showed that we fixed this performance problem. Our process here is noteworthy in a sense that a reproducible benchmark design like in our case depicted in Fig. 2 supports developers to find performance issues before deploying an application to production.

5.2 Concurrency Best Practices

When using MS, one of the greatest challenges is the issue of concurrency control. Therefore, in **RQ2** we ask the question *how can we achieve concurrency control for a mutable data model[...]*? In this Section, we want to address this question and share best practices we identified when implementing the WSS application.

For an immutable data model like BSPD, the concurrency issue is reduced to a minimum since immutable data is inherently thread-safe. We assume that immutable data models are rarely used in OLTP applications. Therefore, developers must explicitly handle concurrency control in their business code and deal with thread management in Java. From lecturing a bachelor's course on concurrency programming [21][16], we know how challenging it is to implement a thread-safe solution with low-level constructs like the *synchronized* keyword. For the sake of simplicity and extensibility, we suggest centralizing all concurrency logic in a single class. This gives a developer the chance to read all code which changes data concurrently in a single or limited number of files. From a portability investigation [13], we know that the lower the number of locations where source code has to be read or changed, the less error prone is the implementation. In the case of WSS this class is called *DataConsistencyManager*. Another important aspect is to prevent the application from becoming deadlocked. We

[15] https://spboehm.github.io/jpa-microstream-doc/.
[16] https://github.com/johannes-manner/ConcurrencyTopics.

used the SERM notation to derive the sequence and hierarchy of lock objects used in our implementation.

Listing 1.1. Lock granularity best practice for MicroStream's concurrent data access.

```
// method for updating order status and customers
public void deliverOldestOrders(... oldestOrders, ...) {
  synchronized (this.storageManger) {
    for (OrderData order : oldestOrders) {
      ...
      synchronized (customer.getId()) {
        synchronized (order.getId()) {
          ...
    this.storageManger.storeRoot();
  }
}
```

When implementing read or write operations, we used the locks from the independent objects towards the dependent objects (Fig. 1) to build nested concurrency blocks within the code as shown in Listing 1.1. For the granularity of locks, we used the identifier of our business objects, a UUID string which is declared as *final* and does not change its identity. This results in an encapsulated concurrency design since the distinct lock object for each Java object is identical for the whole lifecycle of the object. For operations on collections where we want to update several objects of a collection atomically, we used an additional *collection lock object* like the *stockLock* we implemented in our WSS application. This enabled us to handle our collections in a thread-safe manner. A major limitation is how MS writes data to persistent storage. While a write operation is ongoing, the managed Java object graph cannot be modified from other threads. Therefore, we use another lock object (the *storageManager*) since we can only have a single write operation at a time.

When implementing a custom synchronization solution, testing is of utmost importance. Since a verification of the correctness of a parallel program is difficult, brute force testing is one option to assume thread-safety of an implementation with a certain level of confidence. For this, developers can use frameworks such as jcstress[17]. We implemented a stress test for the most critical concurrent operation, the updating of the product stock quantity in our WSS application.

5.3 Usage Scenarios

RQ3 is concerned with possible usage scenarios for MS. The vendor of MS states on their website, that MS is especially suited for "Micro persistence for microservices & serverless Java functions"[18]. When having microservice principles in mind and considering the decentralized data management aspect, their assessment is comprehensible, but the nature of the data model is important for designing an MS solution. As already indicated by the MS vendor's own demo

[17] https://github.com/openjdk/jcstress.

[18] https://microstream.one/.

application (BSPD), good use cases for MS-based persistence may be scenarios with mostly immutable data models. This eases the concurrency control issues as well as the single writing thread bottleneck. When using JACIS, we experienced certain limitations, namely data model decoupling and performance issues. We therefore think that in its current state, JACIS is not a viable option for resolving the concurrency control issue in the context of MS-based persistence. Developers may alternatively use our best practices for implementing a thread-safe solution. But low-level concurrency programming is difficult to get right [11], which in our opinion will therefore limit the adoption of MS as a solution for data storage. For integrating the solution with other databases or systems, the current version of MS provides support for various storage targets, but these adapters often do not support the actual data model of those databases. For example, this means that while MS supports certain relational DBMSs as storage targets, the data stored in these targets by MS is not written as relational data. Additionally, a generic CSV export is offered for data migration. We assume to see more adapters and features with future MS releases, which may also support migration to data models of other persistence solution. This may in turn prove beneficial for the adoption of MS as a persistence solution.

5.4 Threats to Validity

During our comparison of MS and JPA, we had to make choices regarding aspects such as the amount of data used by our benchmark applications, or the execution duration of our benchmark runs. It must be assumed that these choices had an impact on the performance of the systems and therefore, our conclusions. The following listing contains the most important threats to validity from our point of view:

No Lazy References - MS offers lazy references with a semantic similar to JPA's LAZY fetch type for loading data at a later point in time (on demand) which introduces delays since the data is read from disk. For our WSS demo application, we decided not to use this feature since we were able to maintain the entire model data in RAM.

Custom Benchmark Application - We implemented a custom benchmark application and used the BSPD application for reproducing the speedup factor. Although the WSS application is self-audited due to tests, unidentified issues and bugs may still remain. Other applications might face different speedups or even slowdowns. Therefore, the applicability of the results of this work are most likely limited to the current capabilities of the data engine within the context of our modernized implementation of a well-known specification-based benchmark (TPC-C).

Used Experimental Setup - The machines used for our experiments obviously had an impact on the performance of our applications. This might have led to situations where the current experimenter hardware may have favoured one storage approach over the other (disk-based vs. in-memory). In the case of MS, as mentioned in Sect. 5.2, only a single thread can write to disk, as MS will otherwise recognize that parts of the object graph are being modified concurrently and will throw an exception. During the development phase of our test environment, when executing concurrency tests, we faced the situation that disk IO was at maximum capacity when writing the changes, whereas CPU utilization peaked at around 25%. Therefore, the bottleneck in this scenario might have been the disk IO capabilities. Furthermore, while assessing **RQ1**, we were unable to find the hardware configuration used by MS to run their MS version.

6 Conclusion and Future Work

In this paper, we performed a comparison of MS and JPA. First, we evaluated the claims of the MS vendor about the performance superiority of their product over JPA-based solutions. Secondly, we implemented a custom benchmark with a mutable data model, a typical OLTP use case. For this implementation, we found that the MS-based solution does indeed exhibit performance superior to that of a JPA-based approach. When looking at the SPT of the evaluated business function only, in the best case MS was able to outperform JPA by the factor of 400. However, looking at URT, we only observed a speedup of no more than 47%. While this is far from the promise made by the MS vendor, the speedup may still be relevant for latency-critical systems.

For future work, we have three aspects in mind. First, we want to investigate major factors influencing the response time. An abstract model of these factors should include aspects such as payload size, its serialization, and overall HTTP message size. Secondly, we want to compare MS with other in-memory database engines. Lastly, the machines where the benchmarks are executed directly influence the results. Therefore, we want to implement a tool to detect bottlenecks for different hardware configurations based on the benchmarked application. The insights gained in this process can lead to an abstraction from the hardware used. This can help to decompose a machine in relevant components like the CPU, memory, disk, network IO, etc. to build a machine configuration meta model for benchmarks.

References

1. Anderson, T.L., Berre, A.J., Mallison, M., Porter, H.H., Schneider, B.: The HyperModel benchmark. In: Bancilhon, F., Thanos, C., Tsichritzis, D. (eds.) EDBT 1990. LNCS, vol. 416, pp. 317–331. Springer, Heidelberg (1990). https://doi.org/10.1007/BFb0022180
2. Bitton, D., et al.: Benchmarking database systems - a systematic approach. Technical report, University of Wisconsin-Madison, Department of Computer Sciences (1983)
3. Blackburn, S.M., et al.: Wake up and smell the coffee: evaluation methodology for the 21st century. Commun. ACM **51**(8), 83–89 (2008)
4. Boicea, A., et al.: MongoDB vs Oracle - database comparison. In: Proceedings of EIDWT. IEEE (2012)

5. Carey, M.J., et al.: The OO7 benchmark. ACM SIGMOD Rec. **22**(2), 12–21 (1993)
6. Carey, M.J., et al.: A status report on the OO7 OODBMS benchmarking effort. In: Proceedings of OOPSLA (1994)
7. Cattell, R.G.G., Skeen, J.: Objects operations benchmark. ACM Trans. Database Syst. **17**(1), 1–31 (1992)
8. Cooper, B.F., et al.: Benchmarking cloud serving systems with YCSB. In: Proceedings of SoCC (2010)
9. Daynes, L., Czajkowski, G.: High-performance, space-efficient, automated object locking. In: Proceedings of ICDE (2001)
10. DeWitt, D.J., et al.: Parallelizing OODBMS traversals: a performance evaluation. VLDB J. Int. J. Very Large Data Bases **5**(1), 3–18 (1996)
11. Goetz, B., et al.: Java Concurrency in Practice. Pearson Education (2006)
12. Gray, J.N., et al.: Granularity of locks in a shared data base. In: Proceedings of VLDB (1975)
13. Hartauer, R., et al.: Cloud function lifecycle considerations for portability in function as a service. In: Proceedings of CLOSER (2022)
14. Hecht, R., Jablonski, S.: NoSQL evaluation: a use case oriented survey. In: 2011 International Conference on Cloud and Service Computing (2011)
15. Huppler, K.: The art of building a good benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 18–30. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10424-4_3
16. Jordan, M.: A comparative study of persistence mechanisms for the Java^{TM} platform. Technical report, Sun Microsystems Laboratories (2004)
17. Kounev, S., Lange, K.-D., von Kistowski, J.: Systems Benchmarking: For Scientists and Engineers. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41705-5
18. Larson, P., Levandoski, J.: Modern main-memory database systems. Proc. VLDB Endow. **9**(13), 1609–1610 (2016)
19. Larson, P., et al.: High-performance concurrency control mechanisms for main-memory databases. Proc. VLDB Endow. **5**(4), 298–309 (2011)
20. Lilja, D.J.: Measuring Computer Performance: A Practitioner's Guide. Cambridge University Press, Cambridge (2000)
21. Manner, J., Böhm, S.: Lecture notes: concurrency topics in Java. In: Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, no. 106. Otto-Friedrich-University (2022)
22. Manner, J., Wirtz, G.: Why many benchmarks might be compromised. In: Proceedings of SOSE (2021)
23. Sinz, E.J.: Datenmodellierung im Strukturierten-Entity-Relationship-Modell (SERM). Otto-Friedrich-Universität, Bamberg (1992)
24. van Zyl, P., et al.: Comparing the performance of object databases and ORM tools. In: Proceedings of SAICSIT (2006)