MicroStream

Performance White Paper

Table of Contents

Introduction

You can read the following tagline on the website of MicroStream

> Store Java Object Graphs natively, relieved of heavy-weight DBMS Dependencies. Create ultra-fast In-Memory Database Applications & Microservices with Pure Java.
>
> — MicroStream website

This paper will give the figures that prove that MicroStream is ultra-fast and can store your application data with fewer resources than any other solution.

The paper focuses on 2 cases, uploading data and retrieving data.

The comparison

The following frameworks are compared

- MicroStream v7.0
- JDBC (4)
- Hibernate v5.6.9
- JOOQ v3.16.6

For each case, a Java application, running on JDK 17, is created that performs the same functionality using each framework that is listed above. These applications can probably be tuned to achieve a slightly higher execution rate.

However, the idea is to compare the results of each framework if you have a moderate knowledge of the framework. And some minor improvements could be made but it will never change the order of the frameworks in the comparison as the differences are most of the time very large.

The testing setup

In each case, writing a large amount of data and various query scenarios, are executed on a single machine. These are the specification of the machine that is used, although the actual specifications don’t matter since the frameworks are compared using the same resources.

- Mac OS 12.5
- 2,3 GHz 8-Core Intel Core i9
- 16 GB DDR4 memory
- Applications running using Azul JDK 17.0.1
- Closed all additional applications and disabled network and Wi-Fi to have as less interference as possible.
- The database-centered frameworks are using PostgreSQL 14.5 running within Docker Desktop, assigned 8 CPUs and 4GB of memory.

Uploading data

In most applications, adding or updating data happens continuously in small amounts. A few records a time based on user input or incoming data.

determining the performance difference for a few inserts is difficult and probably hardly noticeable. The example uses a large dataset of about 2.8 million records that consist of all trips performed during a month by the drivers. These data are uploaded to analyse various statistics.

The code used for this case can be found in the write *directory*.

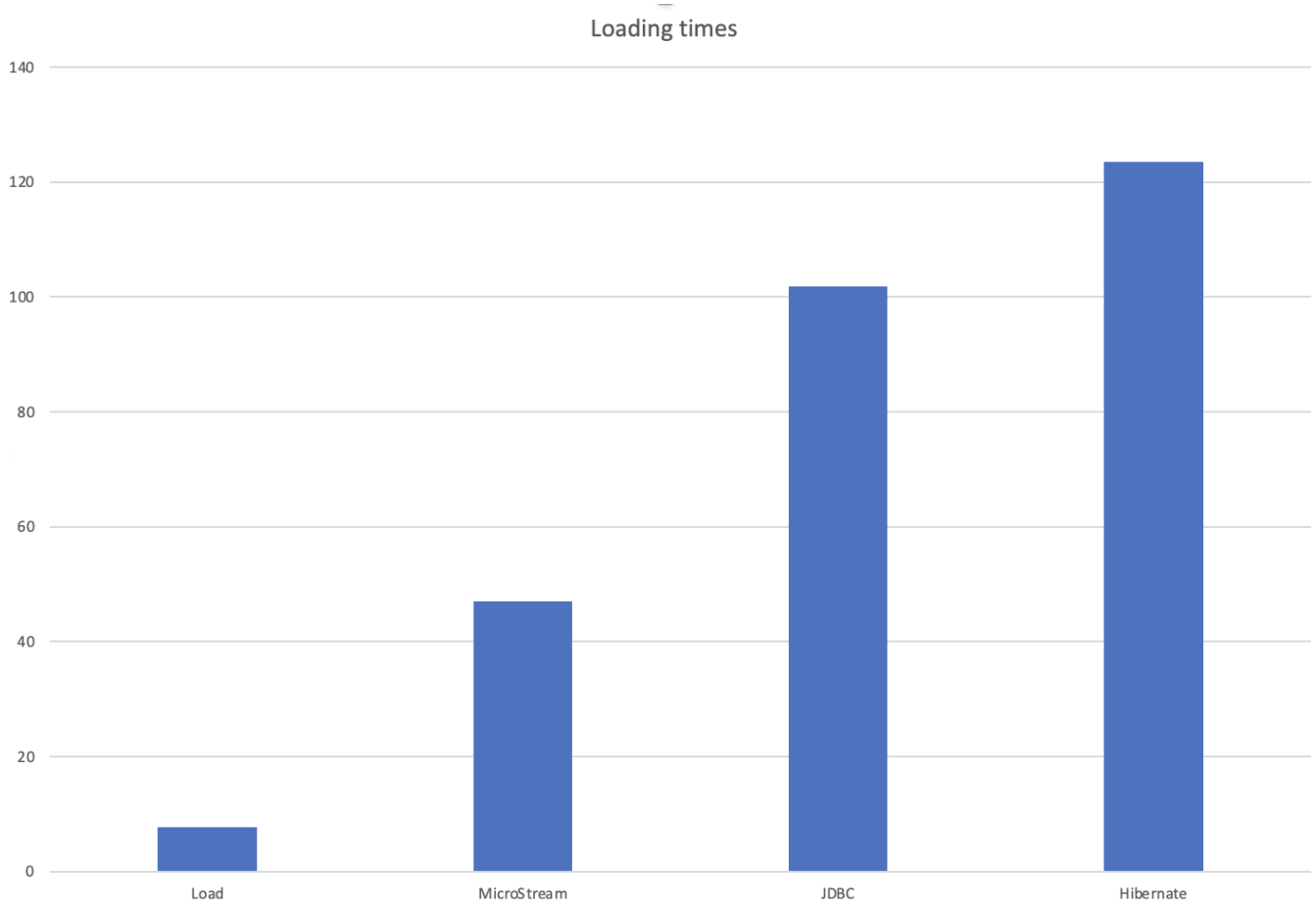
Each framework uses the same pattern for loading data so that they are comparable. After a line is read from the CSV file with all records, a callback method is called that needs to process it in a framework-specific way.

Some specifics

- The MicroStream version places the data in a Map structure where the key is the day of the month and the value is a Lazy list of records for that day. It is the most efficient way to store and retrieve data. Other ways of 'indexing' could be added in a real-world scenario. All data are stored after the loading has finished (day by day)
- The JDBC batching is used to send statements every 1000 inserts to the database. The database commit happens at the end of the program.
- The Hibernate example uses a single transaction and has the option configured to send commands in batch to the database.
- The JOOQ example collects insert statements in a List and sends them in a batch to the database server. However, this option seems to be slow but collecting all statements and sending them all at the end is not possible due to the large memory requirements. This is the reason why the JOOQ example is not on the comparison graphic.

The data consist of 2,821,515 records.

Comparison



Raw data, values are in seconds.

Load	MicroStream	JDBC	Hibernate
7.777	47.000	101.841	123.513

The *Load* value indicates that we can read all records from the data file in about 7.7 seconds.

MicroStream only requires about 40 seconds of additional time to store the data in a binary format to the storage. And we did not need to write some specific code to perform the serialization, nor did we need to set up some external system and prepare this. We just needed to call a single method of the *StorageManager* to accomplish the 'save'.

Both JDBC and Hibernate need much more time to accomplish the same task. They both need about double the amount of time to store them in the database. And as expected, Hibernate requires a bit more time than pure JDBC as the code is more generic and thus less efficient. Both Hibernate provides a better fit within the Object-oriented world and is easier to write and maintain.

Discussion

When playing and analysing the example, we can understand more about the difference between MicroStream and the solutions that use a database.

When you look at the CPU usage, you will see higher utilization of the cores once the data is loaded and we start storing each of the day's data entries.

The example configures multiple channels for the *Storage Manager*. Each channel is a separate thread that performs the conversion and storage of the data to the storage target.

```
channels = Integer.highestOneBit(Runtime.getRuntime().availableProcessors() - 1);
```

This line calculates the number of channels that optimally can be used on the machine.

```
Runtime.getRuntime().availableProcessors()
```

Returns the number of (virtual) cores. We subtract 1 so that we have at least one core available for the other functionality (outside of the channels) and round this down to the highest power of 2 since the number of channels must be a power of 2.

If we change the channel count from 8 (the value calculated for my test machine) to 1, the time to read and store the data increases to 77 seconds. So if you have an application that needs to store large amounts of data in a short period, make sure to increase the channel count to a higher value than the default value 1.

The JDBC and Hibernate programs are comparable to the single channel usage of MicroStream regarding their thread usage. Data is written through a single Session and connection to the database. Instead of serialisation to the binary format of MicroStream, the data is converted to the JDBC data types.

When using Hibernate, the SQL queries for insert need to be generated based on the metadata available on the entity class. This and the other functionality on top of JDBC make that more code is executed than the JDBC example, and thus it is slower.

When the programs send the data to the database, you can see that the CPU usage of the Docker Host is higher than the Java one. The fact that we have an external process to store the data, is an important factor that explains why storing data in a database is much slower than when you use MicroStream. In the example, the CPU usage assigned to the Docker Host process never reached its maximum, so assigning more resources will not increase the performance.

Best practices

One important practice regarding MicroStream is already mentioned in the discussion, the correct assignment of the number of channels.

When your applications need to store a lot of data in a short period, make sure you increase the channel count to 4 or higher. It will improve the throughput when saving the data.

When your application updates and replaces data at a high frequency, make sure that the housekeeping process receives enough time to perform the cleanup. This clean-up is required to remove data from the storage that is no longer needed. Have a look at the Documentation page to increase the time budget assigned to the process.

Conclusion

When storing data, in our example we work with a large dataset so that the differences are very clear, the MicroStream framework achieves much higher performance than using the traditional approach that is based on JDBC and a database.

Not only is it easy to use the full capacity of your machine so that multiple threads write the data concurrently. Also, the absence of an external system that stores your data, the requirement to have data converted in a specific format in the case of a database, latency effects, and additional processing in that external system, make the execution much slower.