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 numbers 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 (4.0)
- Hibernate v5.6.9

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.

Why these frameworks?

When accessing the database, there exist many frameworks, so why did we choose just JDBC and Hibernate?

Some other alternative could be EclipseLink, JOOQ, or QueryDSL. And initially, we added some more frameworks to the comparison but decided to keep only the 2 mentioned because

- During initial tests with these frameworks it was clear that mainly the latency because you use an external system, is the main performance factor.
- Not all frameworks supported the complex queries we used in the examples.

So only JDBC (the bare minimum to use a database) and Hibernate (the most widely used one) are kept.

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 virtual cores)
- 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 taxi drivers. These data are uploaded to analyse various statistics.

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

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

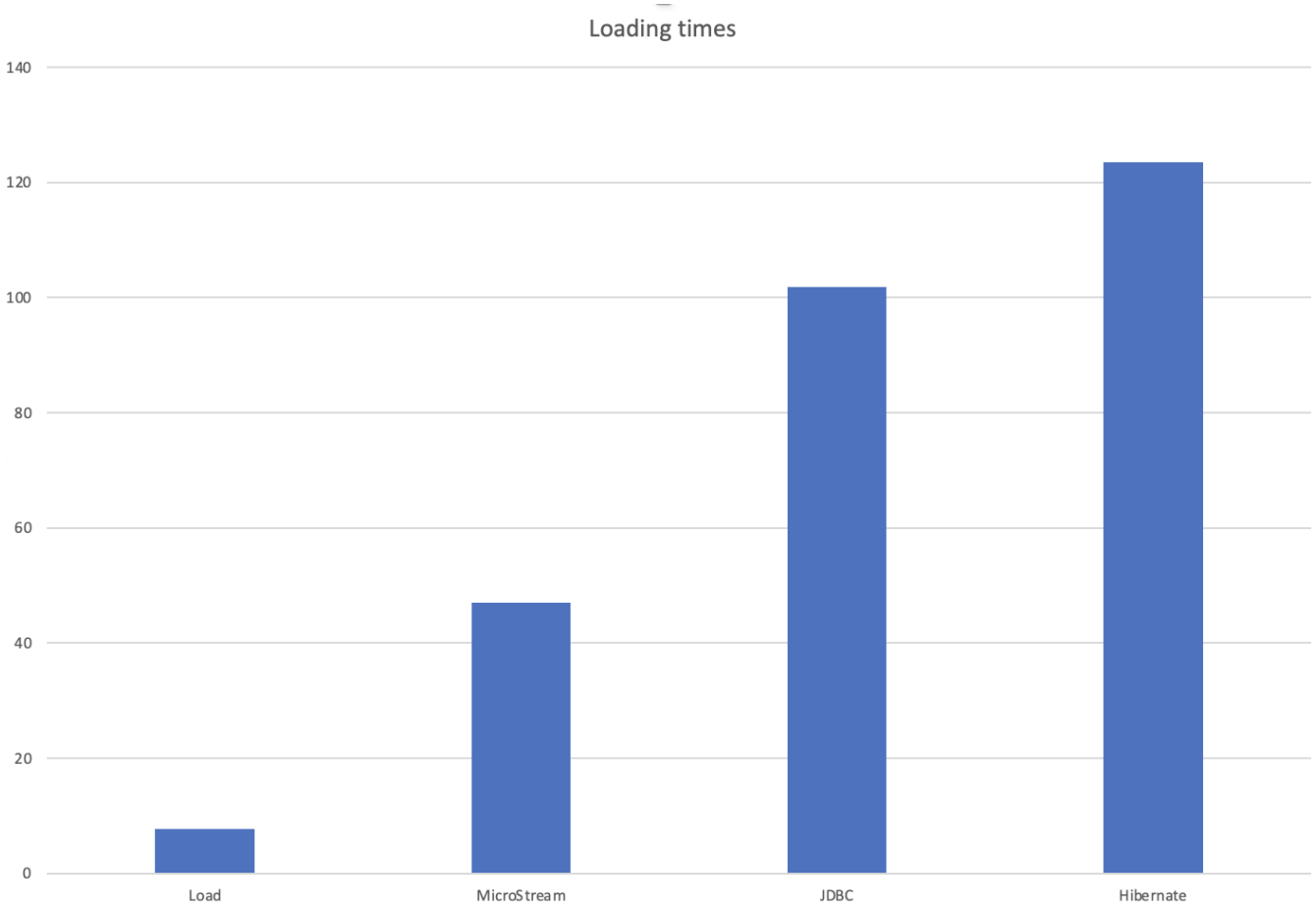
- The MicroStream version places the data in a Map structure where the key is the day of the month and the value is a Lazy list of records for that day. It is the most efficient way to store and retrieve data. Other ways of 'indexing' could be added in a real-world scenario. All data are stored after the loading has finished (day by day)

```
private final Map<Integer, Lazy<List<TripDetailsData>>> tripDetailsByDay;
```

- 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 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 CSV 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. But 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 (previously 47 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.

The (additional) latency that occurs by using an external system, here minimized by running the database on the same machine, and the additional checks the database performs, like the uniqueness check for the primary key, makes the database handling slower than MicroStream.

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.

Querying

Besides the upload, retrieving data is equally important. And in most cases, you query the data more than you upload them, so it might be the most important factor for your application.

Users are in general a bit more patient when they use an application if they know some processing is going on in the background, but data should appear as quickly as possible when they are just browsing through the application.

In the first half of this white paper, we had a closer look at uploading data, now we are going to look at querying the data, and you will see that MicroStream outperforms the frameworks that access a database.

The executing time of some queries is compared that reads data from a book store. The BookStore has shops in several countries. For each shop, we also have the address information (address, city, state, and country) as we have this also for each customer that buys a book in the shop. For each customer, besides address info, we have also the books (s)he bought, the shop in which the purchase was made, and the employee who made the bill. And for each shop, we have an inventory of books available.

Test data for this bookstore can be generated and makes use of the JavaFaker library to have more or less realistic data. There are 3 sizes available for the dataset, medium, large, and huge. For the comparison in this white paper, we have selected the *medium* size.

As a query, we use 2 different examples. One is just a paginated query on the *customer* table. The second one is a much more complex one that involves many tables.

Exploration of the data

Some info about the data volume we have used for this comparison.

- 3 countries
- 43 shops
- 5348 customers
- 200779 purchases over the last 12 years.

The data is generated/uploaded in 2 steps

- Generate the data and save them in the MicroStream format.
- Use these data to upload them to the database tables.

The code used to perform this is located in the classes `GenerateData` and `UploadIntoDatabase` that can be found within the code repository read/generator *module*.

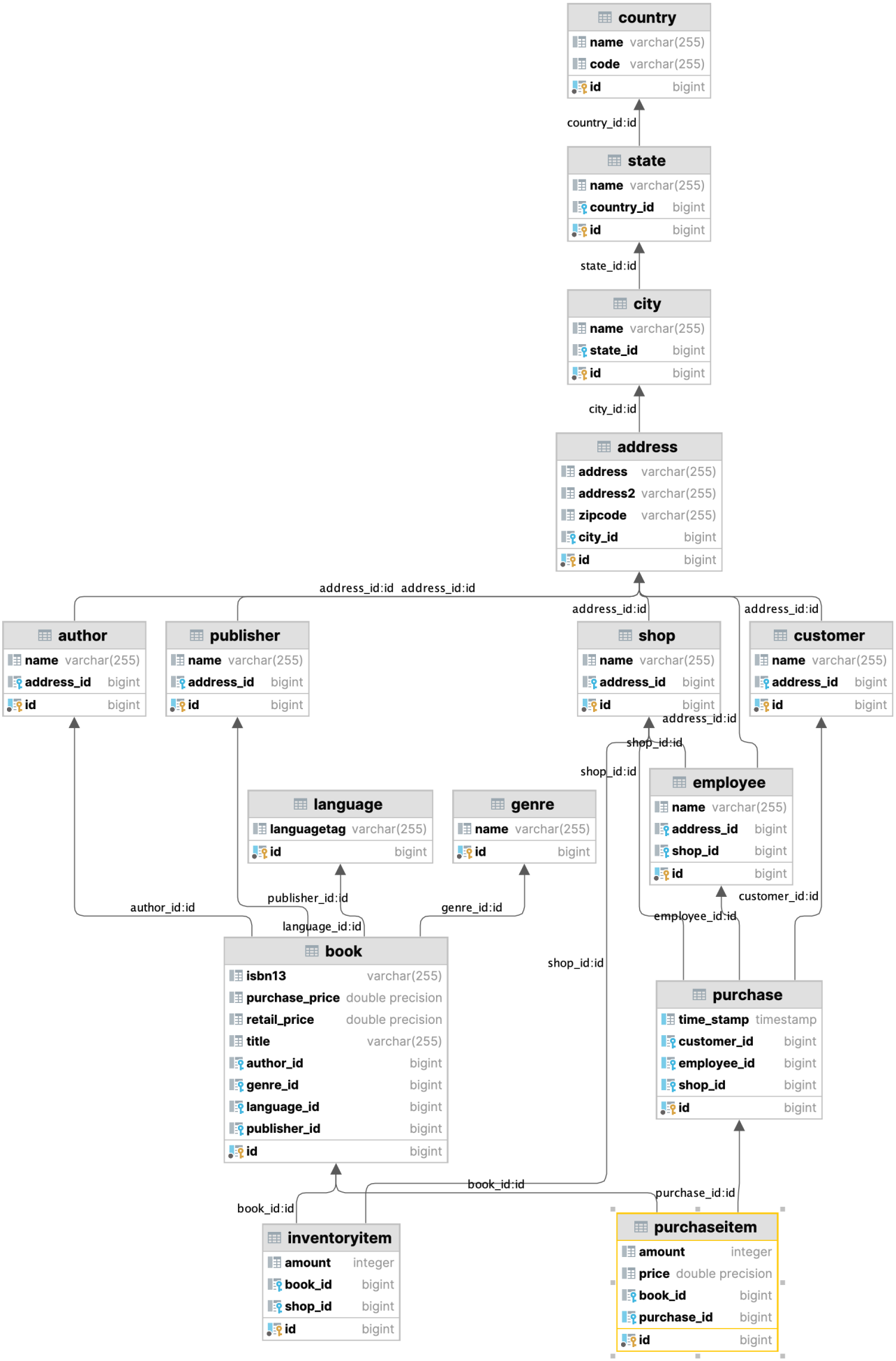
Some specifics

- In the MicroStream case, we only use Lazy references for the purchases. They are organised in a Map.

```
Map<Integer, Lazy<YearlyPurchases>> yearlyPurchases
```

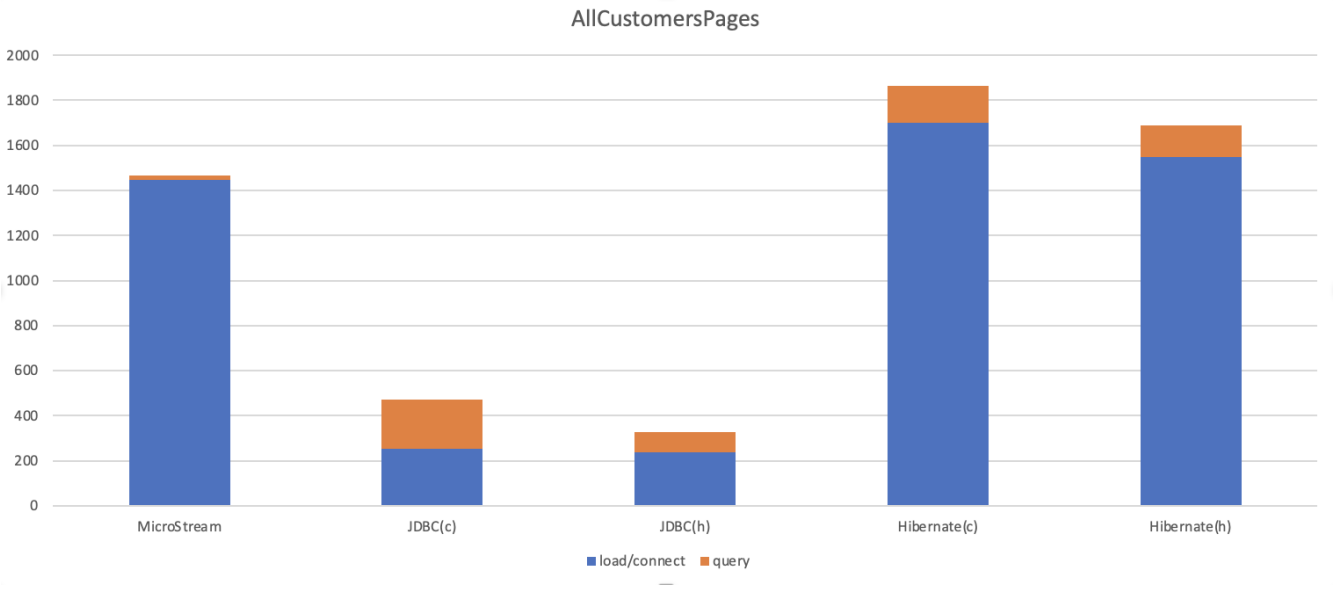
- The class `YearlyPurchases` contains the `Purchases` objects organised in *maps* that are indexed on `Shop`, `Employee`, and `Customer` to have optimal lookup possibilities. All other data is not lazy loaded and read into memory when the *StorageManager* is created.
- The database tables *purchase* and *purchase_item* have database indexes to access the data efficiently. These are the 2 tables with most of the records and can have meaningful indexes to access particular data.
- When using the pure JDBC, a simple mapping framework is used. Since MicroStream and Hibernate return Java classes but JDBC works with sets of *fields*, the simple mapping results also in Java classes. The main goal of this mapping, besides returning objects, is having only a single object for the same record. So there will be only 1 object for a *country*, although it is referenced from different *state* instances.
- The examples using the database are also reading additional tables to correspond to the MicroStream case where you have all referential data available. For example, when retrieving *Customers*, we also retrieve address information (including city, state, and country) as that is probably what is also needed by the application if you need customer information.

The database schema of the book store database looks like this.


country
name varchar(255)
code varchar(255)
id bigint
country_id:id
state
name varchar(255)
country_id bigint
id bigint
state_id:id
city
name varchar(255)
state_id bigint
id bigint
city_id:id
address
address varchar(255)
address2 varchar(255)
zipcode varchar(255)
city_id bigint
id bigint
address_id:id address_id:id
address_id:id
address_id:id
author
name varchar(255)
address_id bigint
id bigint
publisher
name varchar(255)
address_id bigint
id bigint
shop
name varchar(255)
address_id bigint
id bigint
customer
name varchar(255)
address_id bigint
id bigint
address_id:id
shop_id:id
shop_id:id
employee
name varchar(255)
address_id bigint
shop_id bigint
id bigint
language
languagetag varchar(255)
id bigint
genre
name varchar(255)
id bigint
customer_id:id
employee_id:id
author_id:id
publisher_id:id
genre_id:id
language_id:id
book
isbn13 varchar(255)
purchase_price double precision
retail_price double precision
title varchar(255)
author_id bigint
genre_id bigint
language_id bigint
publisher_id bigint
id bigint
shop_id:id
purchase
time_stamp timestamp
customer_id bigint
employee_id bigint
shop_id bigint
id bigint
book_id:id
purchase_id:id
book_id:id
inventoryitem
amount integer
book_id bigint
shop_id bigint
id bigint
purchaseitem
amount integer
price double precision
book_id bigint
purchase_id bigint
id bigint

Comparison Simple Query

As a simple query, we access the *Customer* information using a *paged* approach. There are 3 queries performed, the first one retrieves the first 100 objects (records), the second one the next 100, and so on. The timing is for the 3 queries together.



Raw data, values are in milliseconds.

	MicroStream	JDBC (cold)	JDBC (hot)	Hibernate (cold)	Hibernate (hot)
load/connect	1446	253	239	1702	1554
query	20	217	88	140	132

In the *cold* case, the database was just started and no other action was performed on it yet. In the *hot* case, the timing is performed immediately after the cold run without stopping the database in between.

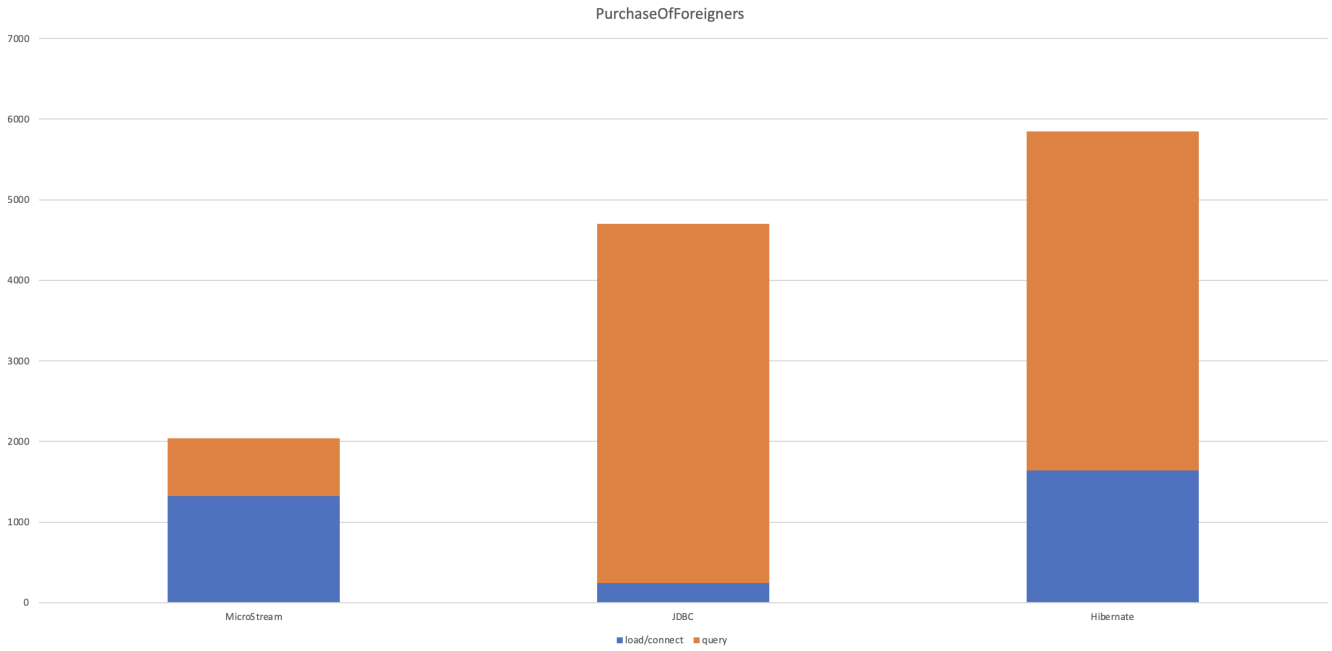
Comparison Complex Query

In the complex example, we retrieve *Purchases* of a certain year when the customer does not live in the same city as the shop where they bought some books.

In the example, 9 queries are executed with 3 different years and 3 different countries (So 3 x 3 is 9 queries)

Current data for JDBC does not contain the purchase detail information, so the required time is larger than presented here!

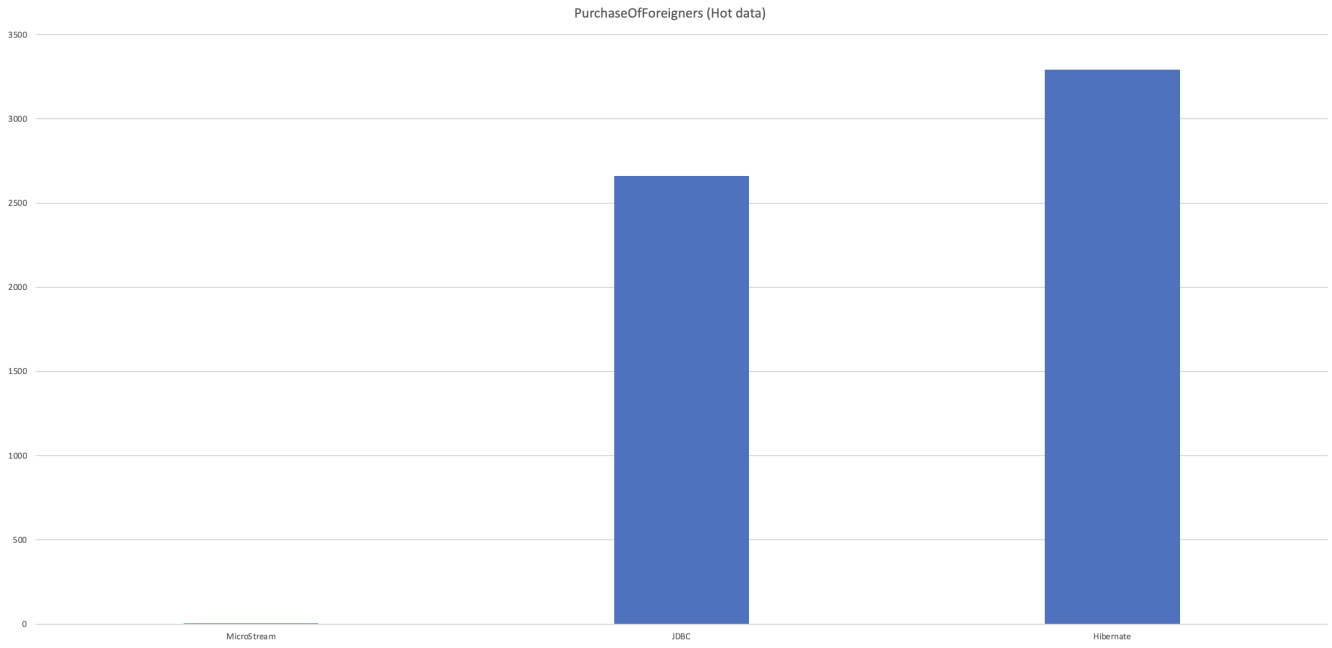
In this *cold* situation, the database was just started and no previous action was launched against it. For MicroStream, it means that the Purchase information is lazy loaded during the *query*.



Raw data, values are in milliseconds.

	MicroStream	JDBC	Hibernate
load/connect	1326	241	1644
query	715	1405	2407

The queries are executed a second time, immediately after the first one, for the same parameters. This means the same connection, and the same Hibernate session is used. For MicroStream, it means that the *purchase* data is already loaded since the same years and countries are used in this second run.



Raw data, values are in milliseconds.

MicroStream	JDBC	Hibernate

5	965	1151

Discussion

When accessing your data, we first need to have a *connection*, and then we can perform the data retrieval. In the case of JDBC, we need to open a JDBC connection to the database. This is relatively fast (around 250 milliseconds in our example)

While Hibernate initialization also requires the database connection, it performs many more tasks. This results in a much longer initialization time of around 1600 milliseconds (1,6 seconds)

This one-time initialization is also required when you use MicroStream. It also loads all non-lazy data at this moment. The time this takes is comparable to the initialization of Hibernate but this time we have already all data (except the purchase data which is loaded lazily) in memory.

Since everything is already in memory, retrieving the customer data in the simple query example is very fast. For MicroStream, it only takes 20 milliseconds to get the data. Also the second time we run the queries, it is rather fast in the case of JDBC. Hibernate retrieves the data in about 140 milliseconds.

The time difference between MicroStream and the database frameworks can be explained by the latency differences. Accessing the data remotely means contacting an external system that needs to interpret the request, fetch the data and return it to the Java process. There we also need to perform a mapping to convert the data into an object structure.

You don’t need any of these actions with MicroStream. The data is already in an object format, you only need to select those instances that fit your criteria.

The same pattern can be seen with the more complex query. Since the condition of the different cities for shops and customers, many tables need to be joined and only then the selection can be made. This means more data need to be handled by the database which makes it takes more time. Also, the fact that each query returns 500 to 1000 records means that more mapping needs to be performed.

In the MicroStream case, the purchase data needs to be lazy loaded but after that, the selection is very fast. You can see that MicroStream can select the purchases in only 5 milliseconds. That is an extremely short time. The first time, when loading the purchase data for the 3 years, the time is just over 700 milliseconds.

But this is still much faster than JDBC and Hibernate can perform the task just by performing the retrieval alone

If we look a bit more in detail at what Hibernate performs when we execute the query related to the purchases of foreigners, we discover that it executes many queries against the database.

Activating the session statistics, it reports that a single query of those 9 queries (the code was adapted to get this information) executes more than 300 JDBC queries. Although the SQL we use explicitly has *JOIN FETCH* statements to make sure we can retrieve the data in one go.

Activating the *show SQL* option reveals that almost all queries are against the author table. And this

is an effect of the Eager mapping we made between Book and Author (when we retrieve a book, you probably want to know the author also) and since the purchase details have the reference to the book, we also read the author information.

When changing this relation to Lazy, we can achieve the execution in 3,9 seconds (or 3,1 seconds in the *hot* case) instead of 4,7 seconds. This is of course an improvement, but still, it is rather slow.

And the answer comes from the JDBC example. When we look at the time that we need to perform the JDBC queries is around 2 seconds (look at the values for the second run. Also when we disable the custom mapping we have written, timings don't vary much). This means the database here is the slowest factor within the example. Performing the queries and fetching the data to the Java program takes about 2 seconds. This is about 220 ms per query (since we executed 9 of them) whereas the simple query example only required 70 ms for a single query.

Best Practices

- It is important that you carefully think about the data model you use with MicroStream. Index alike structures and fast lookups can be achieved by having the appropriate map-like structures and `Lazy` constructs.
- `Lazy` is needed when you have a large dataset and not all data can fit into memory. Lazy is also a perfect match for historic data that you do not always need to be available in the memory.
- `Lazy` also adds additional overhead to the memory (around 1kb) and thus only uses it in a 'granular' way like the example where we have used 1 Lazy object to keep all purchases of a certain year.
- Also each instance that needs to be tracked by MicroStream results in a small overhead. So if you have a root object with many small instances, you should consider using a serialised form of objects (like a JSON string that represents the entire purchase data including referential data) so that the overhead is lower.

Conclusion

Reading data is much more efficient with MicroStream than when you retrieve them from the database. The initial load takes some time, but here in our example it was as fast as initializing Hibernate and from that point on, it can be a factor of 1000 times faster since data is already in the memory and within the correct format.

With a carefully designed data model using Lazy loading, and indexing based on Map structures, you can achieve data retrieval times that are significantly faster. And you do not have the master the complex concepts of database access frameworks, nor spend a lot of time optimizing the database queries.