Mittuniversitetet
MID SWEDEN UNIVERSITY

An Evaluation of Spring WebFlux

With focus on built in SQL features

Karl Dahlin

Thesis - Institution of Information Systems and Technology
Main field of study: Computer Engineering
Credits: 300
Semester, year: Spring, 2020
Supervisor: Johannes Lindén, Johannes.Linden@miun.se
Examiner: Tingting Zhang, tingting.zhang@miun.se
Course code/registration number: DT005A
Degree programme: Master of Science in Engineering Main field of study: Computer Engineering

Abstract

In today's society the need for more hardware efficient software since some people think that the doubling of computer power for the same price that Moore's law predicted is no more. Reactive programming can be a step in the right direction, this has led to an increase in interest in reactive programming. This object of this thesis is to evaluate the possibility of using reactive programming and R2DBC in Java to communicate with a relation database. This has been done by creating two Spring applications one using the standards JDBC and servlet stack and one using R2DBC and the reactive stack. Then connecting them to a MySQL database and selecting and inserting values in to and from it and measuring the CPU usage, memory usage and execution time. In addition to this the possibilities to handle BLOBs in a good enough way were researched. The study shows that there are both advantages and disadvantages with using R2DBC it has basic support and it is based on good idea but at the time of this thesis it still needs more development before it can be used fully.

Keywords: Reactive programming, SQL, database, Java, Spring, JDBC, R2DBC.

Acknowledgments

I would like to thank Easit for supplying a laptop to work on, office space to work in, an assignment to solve, SQL database to work with and guidance in certain areas during the work.

Table of Contents

Terminology

Abbreviation	Description
SQL	Structured Query Language, programming language for interacting with a relation database.
BLOB	Binary Large OBject, a data type that can be stored in a SQL database.
JDBC	Java Database Connectivity, SQL database driver specification
R2DBC	Reactive Relational Database Connectivity, reactive SQL database driver specification

1. Introduction

More and more data is saved and accessed and there is no indication of that trend turning, it is more likely that it will increase in the future since there is continued development of technologies that generate data, for example the Internet of Things and other smart devices.

Java is a programming language introduces in 1995 and it is currently (February 2020) ranked first on *TIOBE's* index of programming languages,the *TIOBE* rankings are based on the number of skilled engineers world-wide, courses and third-party vendors, popular search engines are used to calculate the ratings.[1] *TIOBE* is a company specialized in tracking the quality of software, the quality is measured by applying widely accepted coding standards and *TIOBE* checks more than 1056 million lines of code in real-time every day. [2]

Javas *TIOBE* ranking indicates that Java is a popular programming language and Java is still a growing programming language. There are a lot of features that have come to java over the years and more are in development. Note: Java lost the first place on the *TIOBE's* index of programming languages to C in May of 2020

The interest for more hardware efficient programs and technologies have increased A cause of this might be that people start to doubt that that *Moore's law* will hold up going forward.[3] *Moore's law* basically means that a computers processing power doubles every other year but the cost, power consumption and size stay the same.[4]

An idea for more hardware efficient programs is reactive applications, the concept of reactive programming is not new it has been around for quite some time but it has only been used by a small group of reactive programmers and academics until recently. Observables and Rx almost became buzz words.[5] Reactive application ”react” to changes, a spreadsheet is a great example of this where cells dependant on other cells automatically changes when a change to the cell they depend on occur.[6]

1.1 Background and problem motivation

There is a big, general shift towards asynchronous, non-blocking concurrency in applications. Traditionally, Java used thread pools for the concurrent execution of blocking, I/O bound operations (e.g. making remote calls).

This seems simple on the surface, but it can be deceptively complex for two reasons: One, it's hard to make applications work correctly when you're wrestling with synchronization and shared structures.

Two, it's hard to scale efficiently when every blocking operation requires an extra thread to just sit around and wait, and you're at the mercy of latency outside your control (e.g. slow remote clients and services).

In an ideal world nothing in the stack of an application would be blocking. If an application is fully non-blocking it can scale with a small, fixed number of threads. Node.js is proof you can scale with a single thread only.

In Java we don't have to be limited to one thread so we can fire enough threads to keep all cores busy. Yet the principle remains – we don't rely on extra threads for higher concurrency.

1.2 Overall aim

The problem I will aim to solve in this project is to decide the benefits of a Java program using *R2DBC* and a reactive stack compared to a Java program using *JDBC* and a Servlet stack when working with large result sets.

1.3 Concrete and verifiable goals

The objective of the thesis is:

1. Test the following capabilities of the programs when handling large amounts of data:
   1.1. CPU consumption of the programs.
   1.2. Memory usage of the programs.
   1.3. Measure the execution time.
2. Test if it is possible to handle large BLOBs in a reactive program.
3. Evaluate and present the result from the second, third goal and potential other observations made during the tests and draw conclusions.

1.4 Scope

This thesis will only consider the communication between a Java program and a SQL database and not the rest of the changes that need to be made to a Java program when rewriting it from using a servlet stack to a reactive stack. It will also not test all the different ways of communicating with a database.

1.5 Outline

Chapter 2 describes the relevant theory to the thesis. Chapter 3 describes the method that was used during the thesis. Chapter 4 presents the possible alternatives of frameworks for reactive Java, the alternatives for SQL database and describes the chosen ones and why they were chosen. Chapter 5 describes how the different programs were implemented, the hardware and how the measurements were conducted. Chapter 6 presents the results from the tests. Chapter 7 presents the conclusion based on the result, discussion of the result, discussion of ethical aspects and presentations of suggestions for future work.

1.6 Contributions

Rasmus Holm a fellow student constructed an unofficial LaTeX template that was used when writing the report.

2. Theory

In this chapter information about subjects relevant to the study will be presented and explained briefly. In addition some related work will be presented.

2.1 Spring Framework

The *Spring Framework* was created in 2003 by *Rod Johnson* and it was a response to the complexity that the *J2EE* specifications had at the time.

The *Spring Framework* integrates several technologies, such as Servlet API, WebSocket API, concurrency utilities, JSON Binding API, bean validation, JPA, JMS, and JTA/JCA and it also supports the dependency injection and common annotation specifications that make development easier.

The principles behind *Spring Framework* is:

- Provide choice at every level. Spring lets you defer design decisions as late as possible.
- Accommodate diverse perspectives. Spring embraces flexibility and is not opinionated about how things should be done.
- Maintain strong backward compatibility. Spring's evolution has been carefully managed to force few breaking changes between versions.
- Care about API design. The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.
- Set high standards for code quality. The Spring Framework puts a strong emphasis on meaningful, current, and accurate Javadocs.

Spring is not invasive and makes your application enterprise ready; but you need to help it by adding a configuration to wire up all dependencies and inject what's needed to create *Spring beans* to execute your application.It has two tracks one is *Spring Web MVC* and the other *Spring Webflux*. [7]

2.2 Spring Boot

Spring Boot is a simplified way to create *Spring* applications and an extension to *Spring* and not meant to replace it since it is *Spring*. One of *Spring Boot's* most important features is an opinionated runtime, which helps you follow the best practices for creating robust, extensible, and scalable Spring applications.[7]

There are a lot of configuration needed to run a *Spring* application but *Spring Boot* auto configures these settings. To run *Spring Boot* three things are needed[7]:

- A build/dependency management tool.
- The right dependency management and plugins within the building tool and the *Spring Boot* plugin, to use *Spring Boot Starters*
- A main class containing the *@SpringBootApplication* annotation and the *SpringApplication.run* statement in the main method.

2.3 Model-view-Controller - MVC

Model-view-Controller (*MVC*) is a software architecture style invented by a Prof. Trygve Reenskaug. The architecture style has three main parts, those are the *Model*, the *View* and the *Controller*.

The *Model* is the unchanging essence of the application. In object-oriented terms, this will consist of the set of classes which model and support the underlying problem, this tends to be stable and should have no knowledge about communication with the outside world.

The *View* or *Views* in plural. Is one or more interfaces with the *Model* for a given situation and version. In object-oriented terms classes which give us "windows" onto the model although *Views* often are graphical the do not have to be. Examples of *Views*:

- The GUI/widget view
- The CLI view
- The API view

Views will know of the models existence and some of its nature. An entry field might display or change an instance variable of some *Model* class somewhere.

The *Controller* lets you manipulate a *View*. *Controllers* have the most knowledge of platforms and operating systems. An over-simplification is that the *Controllers* handles the input and the *Views* the output. Just like *Models* have no knowledge of its *Views*, the *Views* have no knowledge of its *Controllers*.[8]

2.4 Reactive programming

Reactive programming is an approach to programming that is an abstraction on top of imperative systems that allows us to program asynchronous and event-driven use cases without having to think like the computer itself.[6]

The short answer to what reactive-functional programming is solving is concurrency and parallelism. More colloquially, it is solving callback hell, which results from addressing reactive and asynchronous use cases in an imperative way.[6]

Reactive programming is useful in scenarios where[6]:

- You process user events or signal changes.
- Handling latency-bound I/O events.
- Handling events pushed to the application

There are many project specifying what reactive is and how to use it, for example the *The Reactive Manifesto*[9] and *Reactive Streams*[10]

2.5 Structured Query Language - SQL – Database

Structured Query Language (*SQL*) and databases running on it is currently an important foundation technology. *SQL* work with one type of database, called relational databases. When communicating with a database *SQL* is used to create a request the database handles the request and returns something. The process of fetching data from

a database is called a database query, the name comes from this. In addition to querying *SQL* can do so much more such as:

- Data definition
- Data manipulation
- Access control
- Data sharing
- Data integrity

This means that *SQL* is a comprehensive language for communicating with and controlling a database management system.[11]

2.6 Java Database Connectivity – JDBC

Java Database Connectivity (*JDBC*) is a standard. All components and techniques of *JDBC* are embedded and implemented in *JDBC API*. Basically, the *JDBC API* is composed of a set of classes and interfaces used to interact with databases from Java applications. The three main functions of the *JDBC API* are:

- Establish connection between Java application and relation database.
- Build and execute SQL statements.
- Process the result.

Different database vendors provide various *JDBC* drivers to support their database.

There are two major sets of interfaces in the *JDBC API* one for the application developers (driver users) and one lower-level for the driver developers. [12]

2.7 Reactive Relational Database Connectivity - R2BDC

Reactive Relational Database Connectivity (*R2BDC*) is a service-provider interface (*SPI*) that provides reactive programming access

to relational databases from Java or other JVM-based languages, it is currently on version 0.8.1. *R2DBC* is based on *Reactive Streams*[1] to allow non-blocking back-pressure-aware data access.[13]

The goal of *R2DBC* is:

- Enabling Reactive Relational Database Connectivity
- Fitting into Reactive JVM platforms
- Offering Vendor-neutral Access to Standard Features
- Embracing Vendor-specific Features
- Keeping the Focus on SQL
- Keeping It Minimal and Simple
- Providing a Foundation for Tools and Higher-level APIs
- Specifying Requirements Unambiguously

[13]

2.8 Related work

A study described in *The Pursuit of Answers* in the book. [6] describes a study in which *Netty* and *Tomcat* were compared. *RxJava* were used in combination with *Netty* during the study. The study showed that *Netty* and *RxJava* performed better during heavy load.

The study differs from what i have done by using a different tool for reactive Java and a different focus in the blocking vs non-blocking comparison. It measures performance and latency of a web service while this thesis focuses on database communication

The study *SpringBoot 2 performance — servlet stack vs WebFlux reactive stack* by Raj Saxena where he compared the servlet stack vs the reactive stack for a web service and testing performance under high load. He found that the reactive stack performs better but that it has a learning curve. [14]

[1] Reactive streams specification "`https://www.reactive-streams.org/`"

The similarities between this study and mine is that both compare the servlet stack and reactive stack in *Spring*. The difference is that the study focuses on the amount of requests the web service can handle and my thesis focuses on the communication with a database.

3. Methodology

The work I will do during this project is literary studies of different internet sources, Java coding to implement two different Spring boot programs and tests for the programs, evaluation and presentation of the results.

The first weeks will be spent on reading and gathering information on the Spring boot and the other connected subjects.

I will start by implementing two different programs for performance testing. One that uses *Spring Webflux* and a reactive stack and one using *Spring MVC* and the servlet stack.

To solve the first goal which is *"Test the following capabilities of the programs when handling large amounts of data:"* and the test sub-goals *"CPU consumption of the programs Memory usage of the programs, Measure the execution time."* I will run the programs and measure the CPU usage, memory usage and execution time of database commands and save the result. The test will consist of inserting and selecting a large amount of a data units from a SQL database using the features of *Spring Webflux* and *Spring MVC* respectively. The execution time will be measured from then a request is sent to the program until it has communicated with a SQL database and after that returned a response to the request sender. CPU usage will be measured by using a program to record the CPU usage of the process during the tests. The memory usage will be measured using a program to record the memory usage of the process during the tests.

To solve the second goal which is *"Test if it is possible to handle large BLOBs in a reactive program."* I will read documentation concerning the handling of BLOBs for the selected implementation of the R2DBC driver and attempt to add functionality for getting BLOBs in a sustainable way to the application using R2DBC.

To solve the third goal which is *"Evaluate and present the result from the second, third goal and potential other observations made during the tests and draw conclusions."* I will present the result from the second goal in an appropriate way then look at it and try to analyze the result and based

on that analysis draw conclusions about the benefits of using a reactive stack compared to a servlet stack.

4. Choice of solution

When working with reactive Java you need to have both a reactive framework for Java and a reactive database driver. There are several SQL databases and frameworks for reactive Java that one can use, in this thesis the SQL database drivers following JDBC and R2DBC specifications will be used. In the following chapter I will mention a couple of different SQL databases and frameworks for reactive Java and what is the focus of this thesis and why.

4.1 Reactive Java

There are a couple of ways of making Java reactive some of these are presented in this section. But first figure 4.1 show how *Spring* and *RxJava* is connected. *Core* is *Reactor core* which is used by *Spring* and the *Reactive Streams Commons* is a joint research effort for building highly optimized Reactive-Streams compliant operators which both *Reactor* and *RxJava 2* implements.[15]

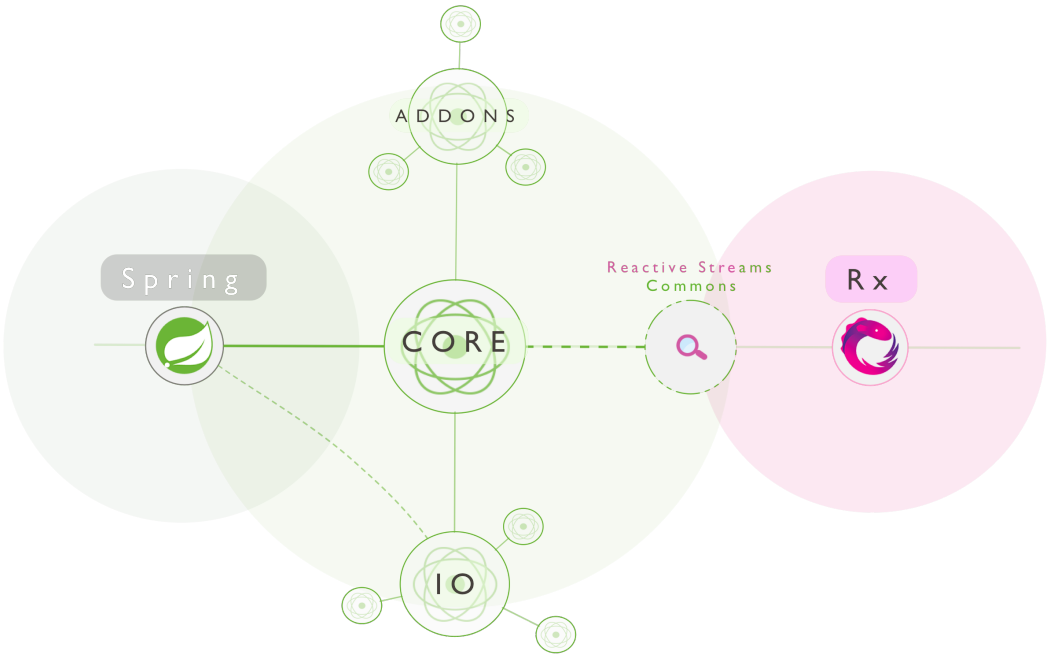

Figure 4.1: Simple overview of how *ReactiveX*, *Reactor* and *Spring* is connected.[16]

4.1.1 Spring webflux

Spring Webflux is a framework for with more then just reactive functionality it is a web framework that runs on server like *Netty*, *Undertow*, and *Servlet 3.1+ containers* and is fully non-blocking and supports back pressure. It uses *Reactor* as reactive library. Since *Spring Webflux* uses *Reactor* as reactive library it has the same base classes:[17]

Class	Description
Mono	data sequences of 0..1
Flux	data sequences of 0..N

Spring Webflux was added in version 5.0 of *Spring*,[17] version 5.0 was released in Sep 2017 which mean that *Spring Webflux* has been released for a little over 2 years at the time this thesis was conducted.

Figure 4.2 shows how the similarities and differences between the reactive stack *Webflux* and the servlet stack *Spring MVC*.

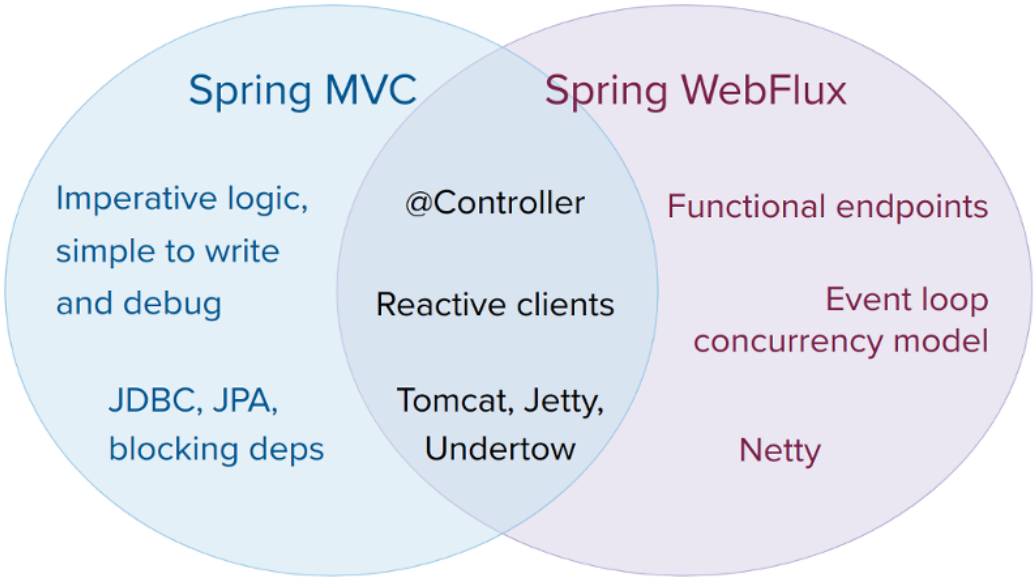

Figure 4.2: Illustation of how *Spring MVC* differs from *Spring Webflux*

4.1.2 RxJava

RxJava makes Java reactive and is a library that implements *ReactiveX* in java and builds on the *Reactive Streams*. RxJava uses the base classes:[18]

Class	Description
Flowable	0..N flows, supporting Reactive-Streams and back-pressure
Obervable	0..N flows, no backpressure
Single	a flow of exactly 1 item or an error
Completable	a flow without items but only a completion or error signal
Maybe	a flow with no items, exactly one item or an error

RxJava is currently on version 3.x, version 1.x reached it's end of life in March 31, 2018 and version 2.x will reach it's end of life in February 28, 2021. [18] *RxJava* version 1.0 were released in Nov 2014.[19] Which means that version 3.x is the current version, version 2.x is reaching it's end of life in about a year and version 1.x reached it's end of life 2 years before this thesis were conducted.

4.2 SQL database

Since JDBC has been around for a longer time then R2DBC and the JDBC driver are more developed only the R2DBC drivers features will be taken into account when deciding on what SQL database to use. The R2DBC is a quite strict specification and it can be found in its entirety here [13]. In this section the implemented features and what is next for the developers of the R2DBC driver is presented, since the R2DBC drivers are continuously being developed the features implemented when this thesis is read may not match what is stated in this section, but this is how it was in mid-February 2020.

4.2.1 PostgreSQL

Driver features of the R2DBC PostgreSQL driver according to the developers[20]:

- Login with username/password (MD5, SASL/SCRAM) or implicit trust
- SCRAM authentication
- Unix Domain Socket transport
- TLS

- Explicit transactions
- Notifications
- Logical Decode
- Binary data transfer
- Execution of prepared statements with bindings
- Execution of batch statements without bindings
- Read and write support for all data types except LOB types (e.g. BLOB, CLOB)
- Fetching of REFCURSOR using io.r2dbc.postgresql.api.RefCursor
- Extension points to register Codecs to handle additional PostgreSQL data types

Their next step is *Multi-dimensional arrays*. [20]

4.2.2 MsSQL

Driver features of the R2DBC MsSQL driver according to the developers [21]:

- Login with username/password with temporary SSL encryption
- Full SSL encryption support (for e.g. Azure usage).
- Transaction Control
- Simple execution of SQL batches (direct and cursored execution)
- Execution of parametrized statements (direct and cursored execution)
- Extensive type support (including TEXT, VARCHAR(MAX), IMAGE,
  VARBINARY(MAX) and national variants, see below for exceptions)

The data types CLOB and BLOB is fully materialized in the client before decoding, their next step is *Execution of stored procedures* and *Add support for TVP and UDTs*. [21]

4.2.3 MySQL

Driver features of the R2DBC MySQL driver according to the developers[22]:

- Unix domain socket.
- Execution of simple or batch statements without bindings.
- Execution of prepared statements with bindings.
- Reactive LOB types (e.g. BLOB, CLOB).
- All charsets from MySQL, like *utf8mb4_0900_ai_ci*, *latin1_general_ci*, *utf32_unicode_520_ci*, etc.
- All authentication types for MySQL, like *caching_sha2_password*, *mysql_native_password*, etc.
- General exceptions of error code and standard SQL state mappings.
- Secure connection with verification (SSL/TLS), auto-select TLS version for community and enterprise editions.
- Transactions with savepoint.
- Native ping command that can be verifying when argument is ValidationDepth.REMOTE

Their next step is *Prepared statements cache* and *Statement parser cache*. [22]

4.3 Chosen solution

Spring webflux was chosen as the framework for making Java reactive since it is a part of the larger *Spring Framework* and they use *Spring Webflux* and *Spring Framework* at Easit and they were the ones that supplied the assignment and the *Spring Framework* is also a large and widely used framework.

MySQL was chosen as the SQL database since it was the only driver that claimed to have reactive support for BLOB and CLOB types and

I want to test how a reactive system handles that as well see section 4.2.3 for all the features of the driver. The *MySQL* driver also had implemented all the support for simple SQL statements, all drivers had this.

5. Implementation

In this chapter the implemented programs and their parts are described. Figure 5.1 shows the planned overall layout of the system where the *Request sender* is a program that sends requests to the different *Spring boot* programs, *Spring boot program* is either the program with servlet stack or the program with reactive stack. When measuring the CPU usage, memory usage and execution time the programs will connect to a remote SQL database, but when this is not the focus of the test some tests will be run with the database running on the same machine as the program. The version of Java that will be used in this thesis is Java 11.

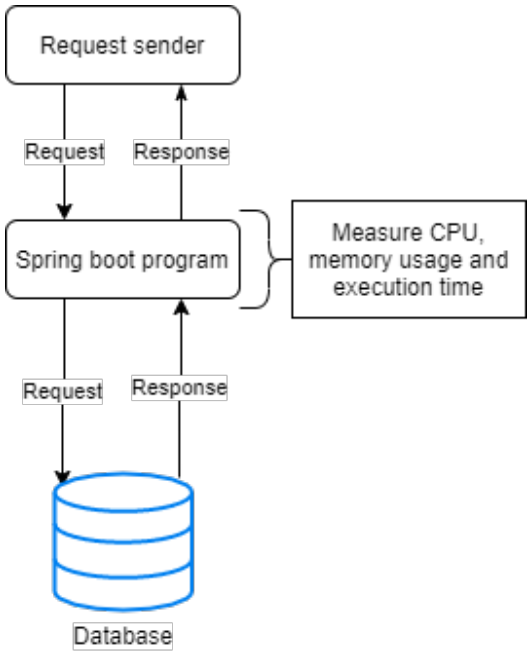

Figure 5.1: Simple overview of the layout of the programs.

5.1 Hardware

In this section the two laptops specs will be presented, the second one is borrowed Easit.

The first laptop is a HP EliteBook 840 G3 with a Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, 2496 Mhz, 2 Core(s), 4 Logical Processor(s) and 8 GB RAM. The laptop runs Windows 10 Pro version 10.0.18363 Build 18363.

The second laptop is a Dell Latitude 5590 with a Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 2112 Mhz, 4 Core(s), 8 Logical Processor(s) and 16 GB RAM. The laptop runs Windows 10 Pro version 10.0.18363 Build 18363.

The first laptop will be referred to as *the first laptop* or *the HP laptop*, the second laptop will be referred to as *the second laptop* or *the Dell laptop*.

The local installation of the MySQL server on the HP laptop is of version 8.0.19 (MySQL Community Server - GPL) and the remote server runs a Mysql server of version 8.0.12 (MySQL Community Server - GPL)

5.2 Programs

In this section the programs that will be created for testing will be described. Both of the programs created to test the database communication will be web clients listening on port 8080 that receives a request from a third party. To enable this there are some things that *Spring boot* auto-configures, these can also be changed if needed. For example the *Spring MVC* program needs a Servlet container and the *Spring Webflux* needs and async and non-blocking server.

5.2.1 Spring MVC program (Servlet)

The reactive program were created using *Spring initializr* with the dependencies *Spring web*, *MySQL Driver*, *Spring Data JPA* and *JDBC API*, some of these might not be needed in the end. The *Spring MVC* program will be developed to get a baseline to compare the result from the *Spring Webflux* program. The program will be created with *Spring boot* to shorten the configuration and enable a quick start when developing.

The program will use the *JDBC* MySQL driver and use the *java.sql.Connection* to communicate with the database. The default embedded Servlet container *Tomcat* will be used.

For inserting values autocommit will be turned off and a *java.sql.PreparedStatement* will be used to add insert commands to its *batch* and when a set number has been added the *batch* will commit to the database and this cycle will repeat until the desired number have been

inserted into the database. The thing that will be counted is the number of column values.

For selection values the fetch size will be set and data will be fetched by executing a *java.sql.Statement* and adding the return result to a *ArrayList* and returning the *ArrayList* to the *Request sender*.

5.2.2 Spring Webflux program (Reactive)

The reactive program were created using *Spring initializr* with the dependencies *Spring Reactive Web*, *MySQL Driver* and *Spring Data R2DBC*. The dependency *R2DBC Proxy* will also be added in later. The dependency *R2DBC Proxy* will be added in to simplify development by enabling query logging and method tracing.

The program will use the community-driven *R2DBC* driver for MySQL, use the default async and non-blocking server *Netty*.

The program will interact with the database using a feature from the *Spring Framework* called *repositories*, a class called *org.springframework .data.r2dbc.core.DatabaseClient* and a class called *io.r2dbc.spi.Batch*.

When it comes to inserting into and selecting from the database this will be done in the methods mentioned above and support for testing what difference the reactive transactions built into the *Spring Framework* makes. In addition to this support for getting *BLOBs* will be implemented and in such a way that the entire *BLOB* wont be loaded into the memory of the application at once.

A client program will be created as one of the *Request senders* since when it comes to the response from the reactive program the response might need to be handled in a reactive way. This will be done by creating another *Spring boot* with just the *Spring Reactive Web* dependency then configuring a *CommandLineRunner* Bean to make a GET request to the reactive program and then handling the response as a *Flux* or *Mono*.

5.3 Measurements

Where the test will be run may wary depending on what is measured if time, CPU usage and memory consumption are measured the test will be run on the Dell laptop unless stated otherwise.

The CPU usage and memory usage measurements will be taken with a program called *VisualVM* is a troubleshooting tool with good features. It used to be a part of Oracle JDK 6-8 as *Java VisualVM* but from JDK 9 it was discontinued and is now a separate program. [23]

Figure 5.2 shows the layout of *VisualVM*, to the left the currently running java processes can be seen in a list and if one is double-clicked the monitoring view for that process pop up in the rest of the window. The currently shown tab is the *Monitor* where you can monitor an application in real-time, in addition to this there are tabs for *Overview*, *Threads*, *Sampler* and *Profiler* and these displays information corresponding to their name. In addition to monitoring resources it is possible to see how long a process has been running for.

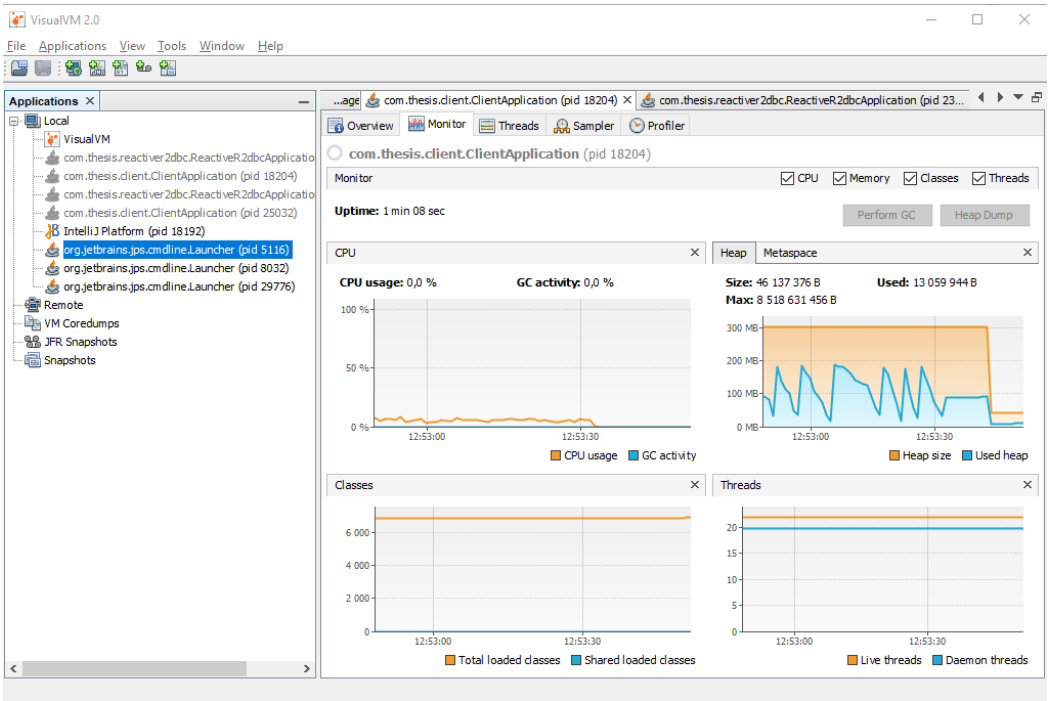

Figure 5.2: Screenshot of one VisualVMs monitoring view.

Execution time will be measured by a program called *Postman* or by logging from the program. Figure 5.3 shows the layout of *Postman* where you can select what type of request to send, where to send it and add params if needed among other things. It also displays the response code, the time it took to send a request and a receive response, the size of the response and the response itself.

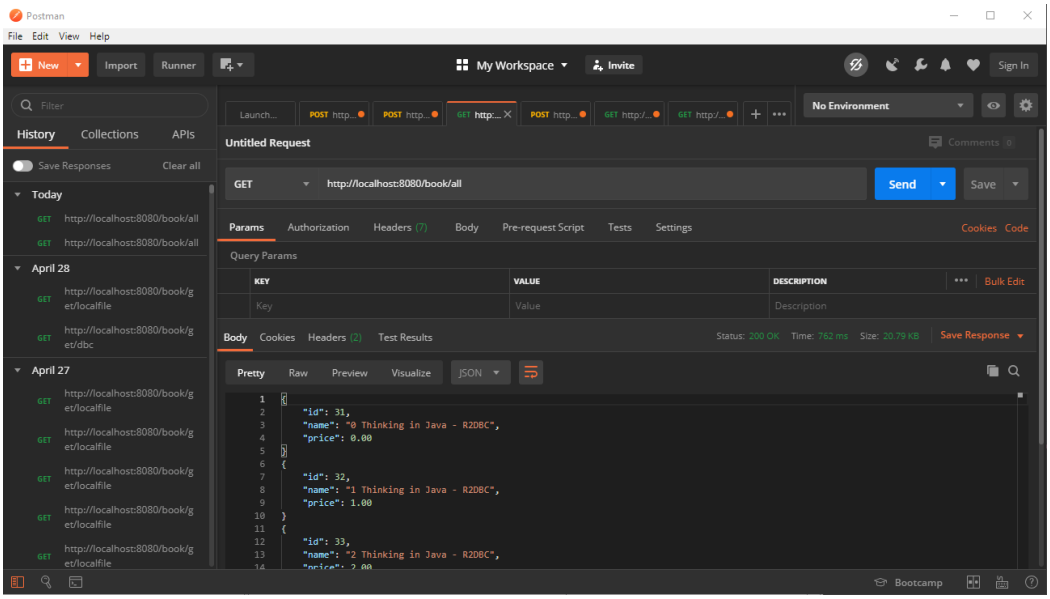

Figure 5.3: Screenshot of Postman

6. Results

In this chapter the features of the programs are explained, the result from the performance tests is presented and the result of the study of the possibility to handle BLOBs in a reactive program is presented.

6.1 Programs

In this section the result of the implementation is presented, the thesis resulted in three different *Spring* programs. One *JDBC* program using servlet stack, one *R2DBC* program using reactive stack and one reactive client that sends a request to the reactive program and receives and handles the response.

6.1.1 JDBC program

The program is a standard *Spring MVC* program that is a web service that runs on the local machine on port 8080 and takes a request processes it and communicates with a SQL database. The program can handle GET and POST web requests that correspond to SELECT and INSERT SQL commands respectively.

To handle INSERTs into the database the program uses a *PreparedStatements* and adds them to a batch this is wrapped in a custom made *BatchHandler* that keeps track of the added number of columns values and when the number of column values equals the batch size the batch is committed.

To handle SELECTs from the database, the program a *JdbcTemplate* is used and created from the connection information to the database the query is executed and the result is processed. With *JdbcTemplate* it is possible to set fetch size which determines the size of each chunk of the reply. For these tests the default is used.

6.1.2 R2DBC program

The program is a standard *Spring Webflux* program that is a web service that runs on the local machine on port 8080 and takes a request processes it and communicates with a SQL database. The program can handle GET and POST web requests that correspond to SELECT and INSERT SQL commands respectively.

Communication with the database will be done in three ways *Repositories* a feature of *Spring data*,*Databaseclient* a class that sends commands to a configured database and *Batch* a class that sends commands to a configured database by a class called *Connection*. The main focus is on the repositories.

When returning data from the database to the request sender it can be done as a stream of data or chunks.

6.1.3 Reactive client

The program is a simple *Spring Webflux* application that is a web service that runs on the local machine on port 8081 and sends a web request to a specified target in this thesis a GET request to the reactive *R2DBC* web service. The program uses a *CommandLineRunner* to send the request on startup, the response is later processed. Since it is a *Webflux* application communicating with another *Webflux* application the returned data will both be sent and processed as either a *Flux* of some type or a *Mono* of some type then printed in the console of the client application.

6.2 Tests

In this section performance tests are described and the result of the tests run on the two *Spring* programs is presented. CPU usage is the amount of CPU the process takes from the system, memory usage is the amount of memory the process takes from the system and the execution time is the time it takes from a program to send a request to the web service, the web service communicate with the database and then return the reply to the program that sent the request.

For the insertion tests the data is generated in the same way so it and the class used in the tests is a custom created class called *Book*. Figure 6.1 shows the Java code for the class where internal variables are present, *id* is an id that is auto-generated by the database when a book is added to it, *name* represents the name of a fake book and *price* represents the price of a fake book.

```java
public class Book {

    @Id
    private Long id;
    private String name;
    private BigDecimal price;

    public Book(Long id, String name, BigDecimal price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public Book( String name, BigDecimal price) {
        this.name = name;
        this.price = price;
    }

    public Book(Book in) {
        this.id = in.id;
        this.name = in.name;
        this.price = in.price;
    }
```

Figure 6.1: Screenshot of the code for the *Book* class.

Figure 6.2 show the layout of the table used in the tests taken from MySQL workbench. Where *id* is an auto-generated id assigned by the database when a value is inserted. Name is a 255 characters long VARCHAR where the name of the fictional book. Price is a 15 characters long NUMERIC where there can be two after the decimal point.

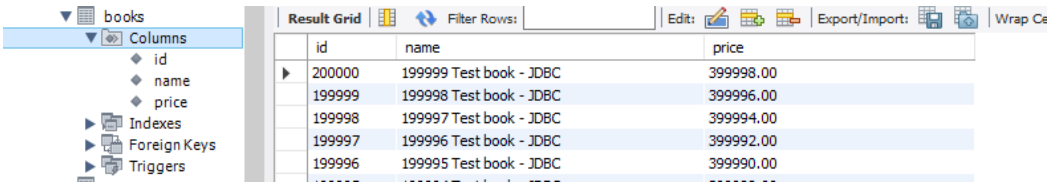

Figure 6.2: Screenshot of the columns in the table used for the test.

The INSERT tests are run on an empty table in the database and the database table is removed and recreated with the SQL commands shown in figure 6.3.

```
1 •  DROP table IF exists books;
2
3 •  CREATE TABLE books(id SERIAL, name VARCHAR(255), price NUMERIC(15, 2));
```

Figure 6.3: Screenshot of the SQL commands for removing and recreating the table.

The SELECT tests are run on a table with 1 000 000 rows with id 1 to 1 000 000.

6.2.1 JDBC program

The first test on this program is an INSERT of 200 000 *Books* using a batch size of 100 column values. Figure 6.4 is a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

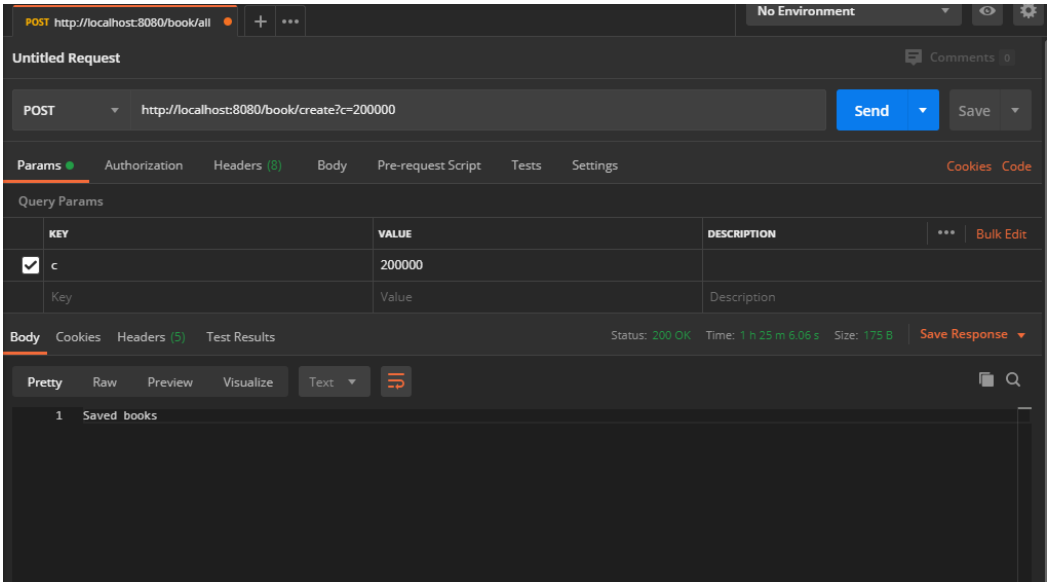

Figure 6.4: Screenshot of Postman after sending command to insert 200 000 values in to the table.

Figure 6.5 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when an INSERT of 200 000 instances of the type *Book* is inserted into the database with batch size 100. It shows the process uptime, CPU usage, RAM usage, loaded classes and

threads.

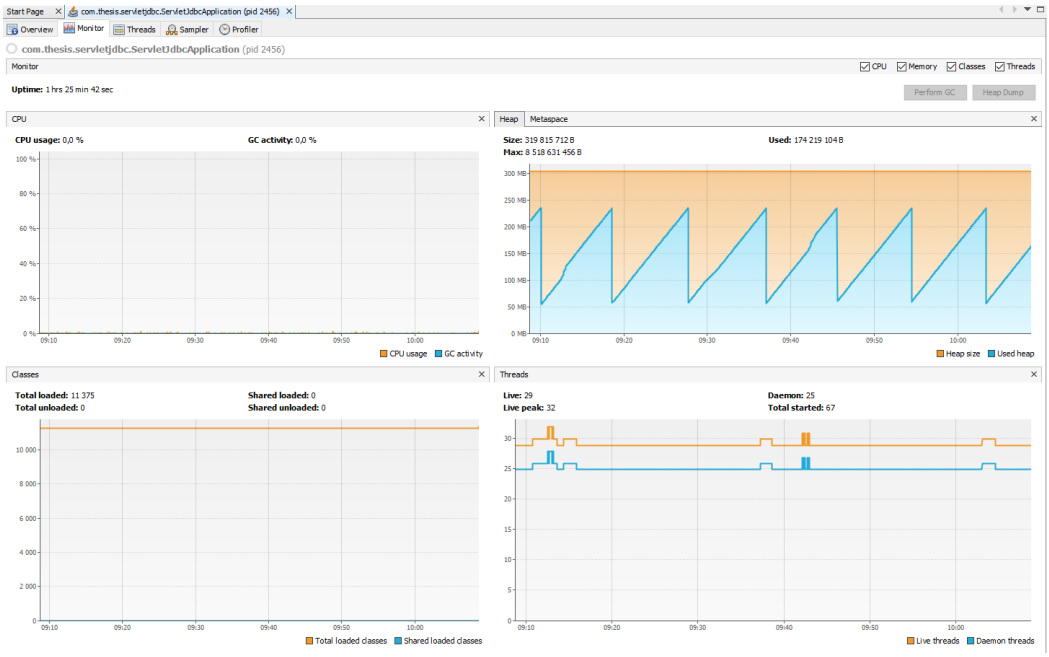

Figure 6.5: Screenshot of VisualVM after sending command to insert 200 000 values in to the table.

The SELECT test was run with 1 000 000 values in the table of the database run a couple of times and always ended up taking around five seconds. Figure 6.6 show a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

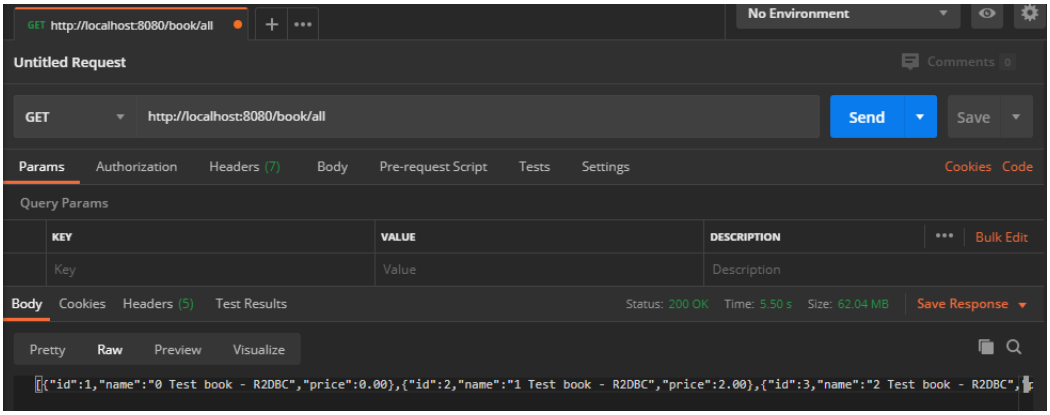

Figure 6.6: Screenshot of Postman after sending select command with 1 000 000 values in to the table.

Figure 6.7 show in more detail where the time of the request were spent.

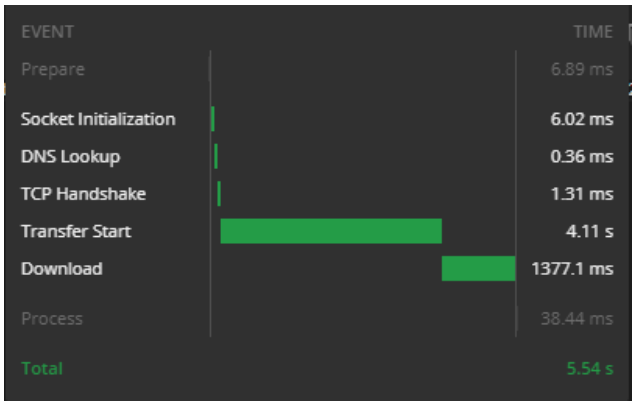

Figure 6.7: Screenshot of time details Postman after sending select command with 1 000 000 values in to the table.

Figure 6.8 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when an INSERT of 200 000 instances of the type *Book* is inserted into the database with batch size 100. It shows the process uptime, CPU usage, RAM usage, loaded classes and threads.

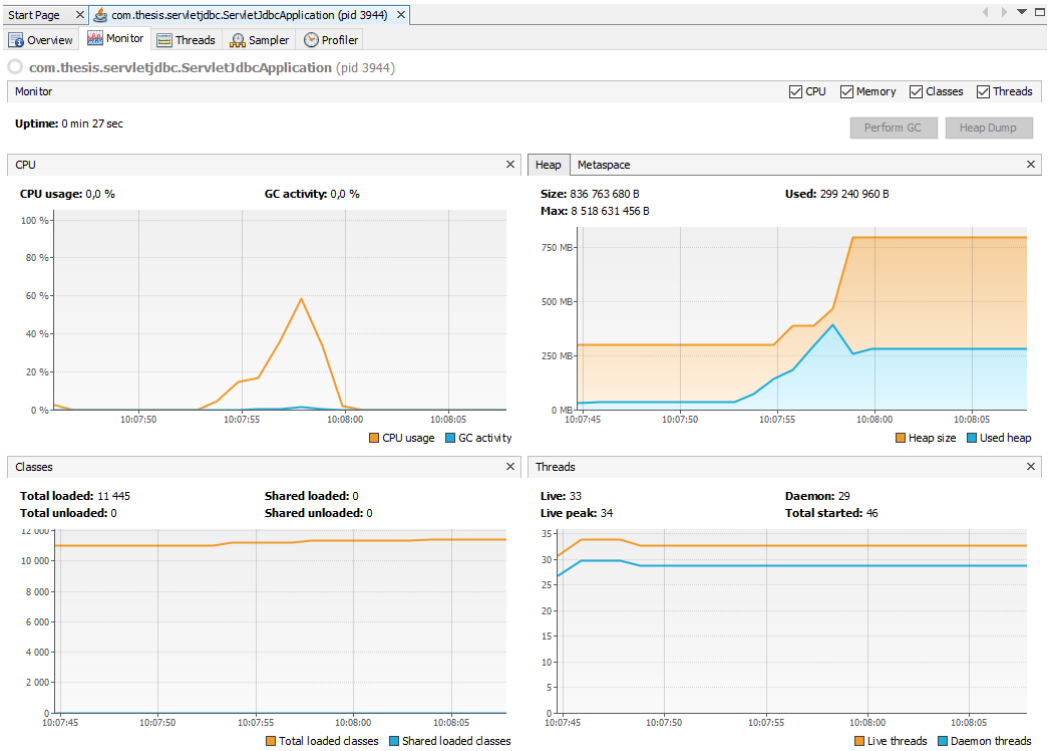

Figure 6.8: Screenshot of VisualVM after sending select command with 1 000 000 values in to the table.

An almost identical test was run but with the difference that the heap space was limited to 256 MB. This resulted in a crash of type *java.lang.OutOfMemoryError: Java heap space*. In figure 6.9 it is visible when the program ran out of heap.

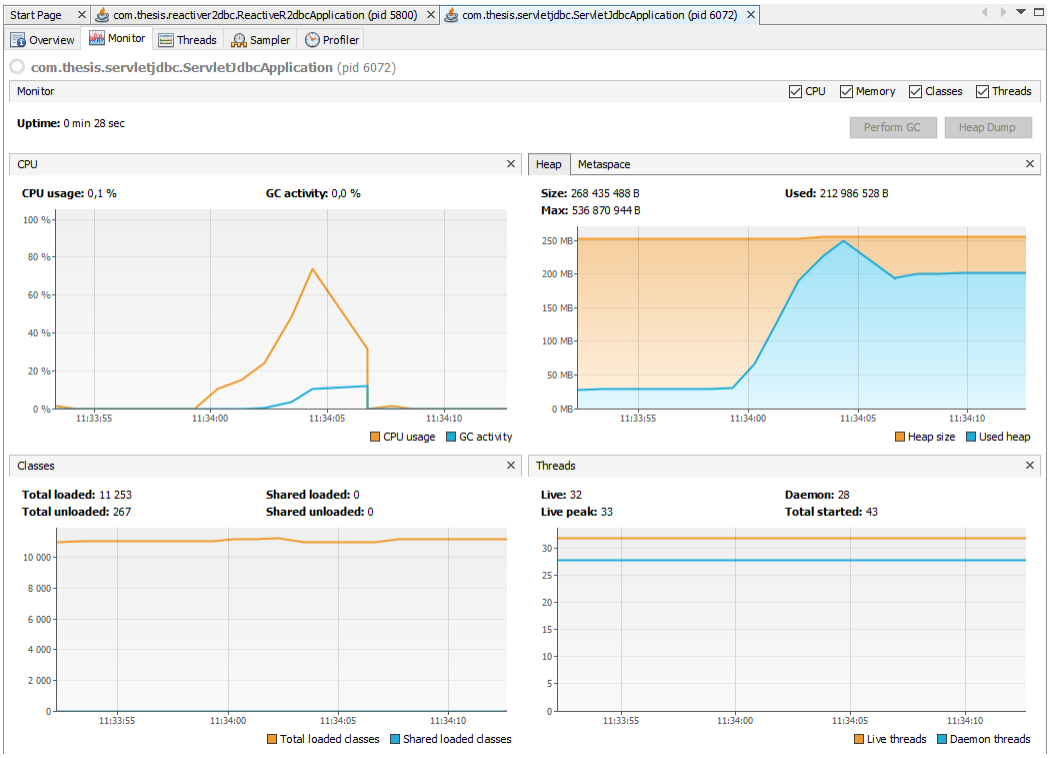

Figure 6.9: Screenshot of VisualVM after sending select command with 1 000 000 values in to the table and limited heap memory.

6.2.2 R2DBC program

The annotation *@Transactional* enables the built-in reactive transactions in *Spring Webflux*, if it is not enabled the default behavior is *auto commit*. So result from runs with both the default and the reactive transaction will be presented in this section.

INSERT

This test on the program is an INSERT of 200 000 *Books* using a *repository* with reactive transactions enabled. Figure 6.10 is a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

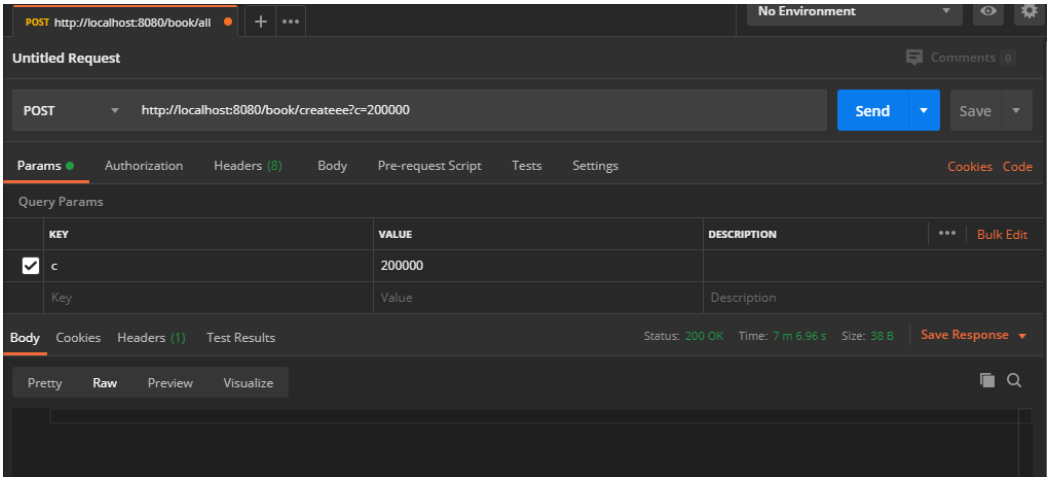

Figure 6.10: Screenshot of Postman after sending command to insert 200 000 values in to the table.

Figure 6.11 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when an INSERT of 200 000 instances of the type *Book* is inserted into the database with a repository with reactive transactions enabled. It shows the process uptime, CPU usage, RAM usage, loaded classes and threads.

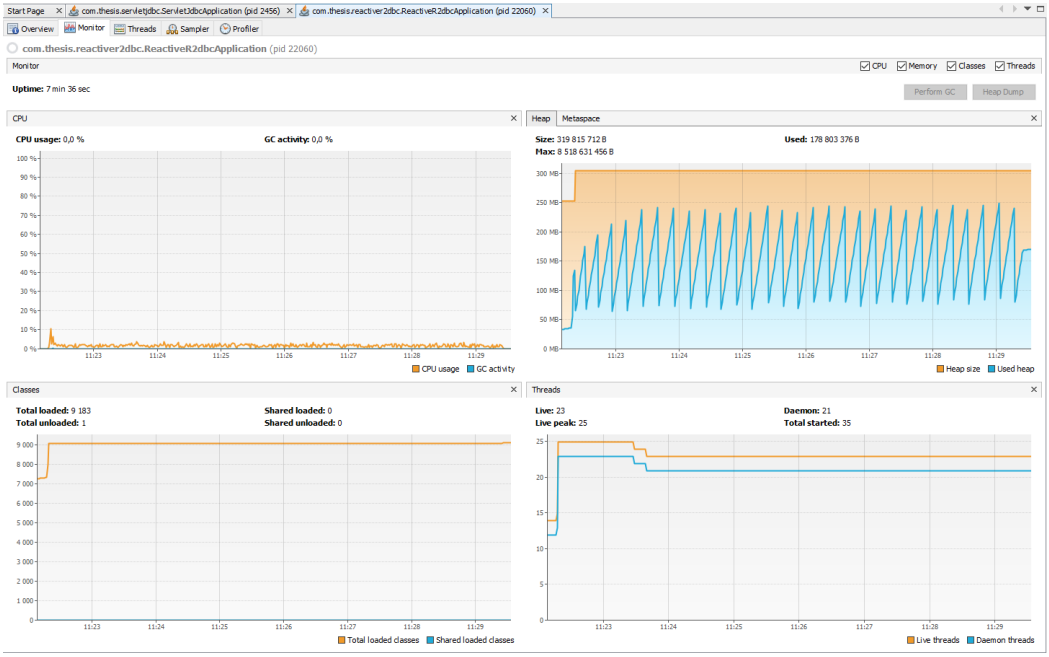

Figure 6.11: Screenshot of VisualVM after sending command to insert 200 000 values in to the table.

This test on the program was supposed to be an INSERT of 200 000 *Books* using a *repository* with reactive transactions disabled. The test was canceled after it had been running for one hour. Figure 6.12 shows the values in the database table after one hour of tests, when the test was canceled.

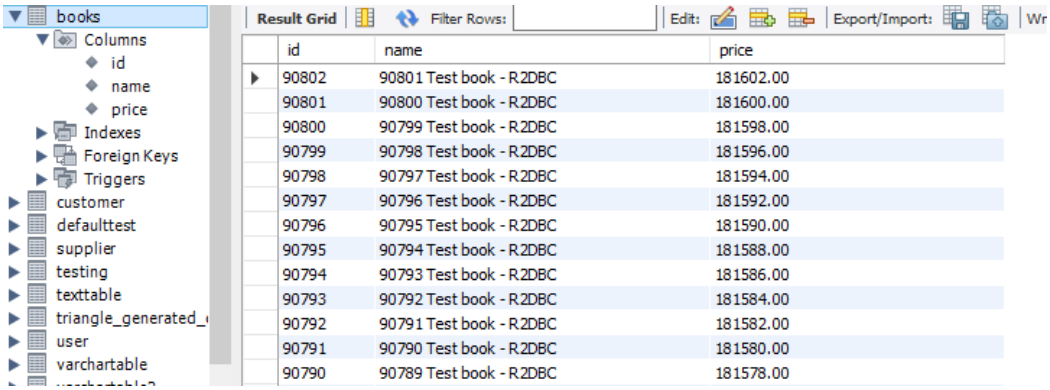

Figure 6.12: Screenshot of MySQL workbench after test with inserts of 200 000 values in to the table had been running for an hour.

Figure 6.13 is a screenshot of *Postman* that shows the type of request and to what address it is sent and no response or response time since the request was canceled.

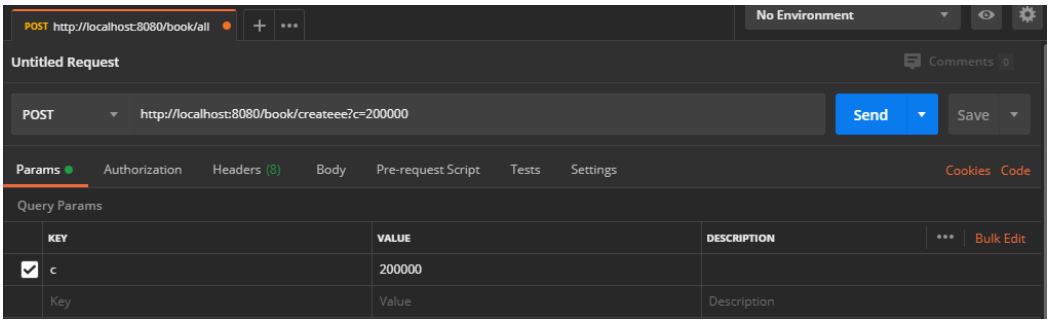

Figure 6.13: Screenshot of Postman after sending command to insert 200 000 values in to the table without the reactive transactions.

Figure 6.14 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when an INSERT of 200 000 instances of the type *Book* is inserted in to the database with a *repository* with reactive transactions disabled. It shows the process uptime, CPU usage,

RAM usage, loaded classes and threads. which shows that the process had been running for one hour and eight minutes when the test was canceled.

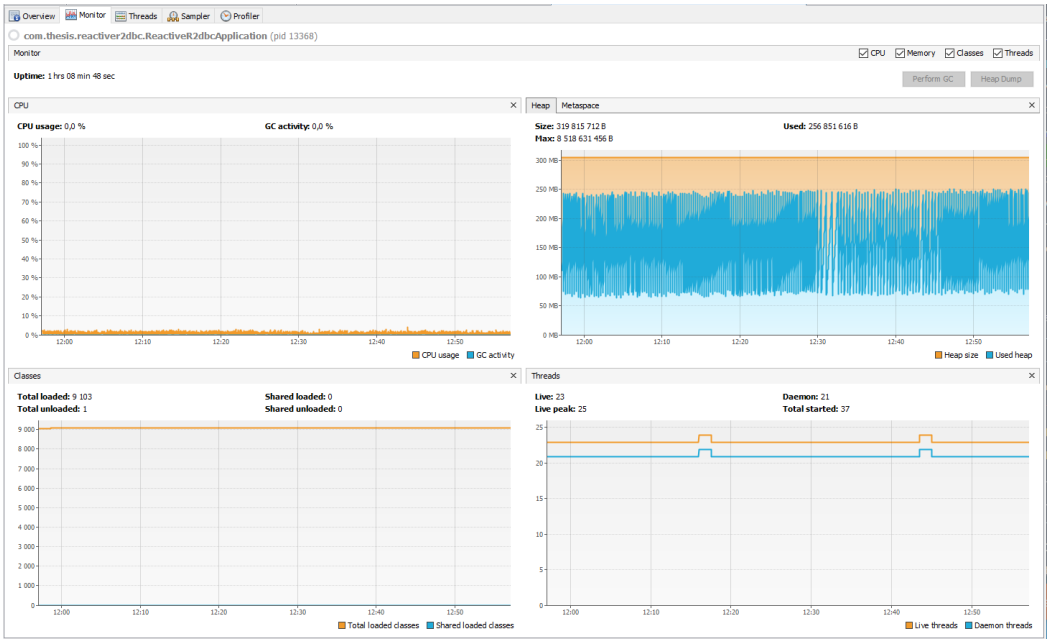

Figure 6.14: Screenshot of VisualVM after sending command to insert 200 000 values in to the table.

Due to the very slow speed of not using reactive transactions the rest of the tests on this program will only be run with reactive transactions enabled.

Tests with the *DatabaseClient* class are executed one after another by using commands like:
databaseClient.execute("INSERT INTO books (name, price) VALUES ("Test Book - R2DBC", 5)")
And this would result in one statement being sent to the database at a time and each statement handled individually and that would result in similar results as the *repository* tests without reactive transactions. So no 200 000 INSERT tests were done with the *DatabaseClient*.

The INSERT test with 200 000 values using *Batch* class failed with the output in figure 6.15. Got a packet bigger than 'max_allowed_bytes'.

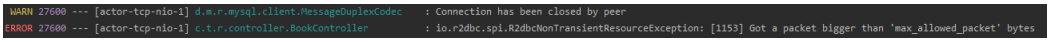

Figure 6.15: Screenshot of the error output when inserting 200 000 values.

Figure 6.16 shows the Postman output of the failed test.

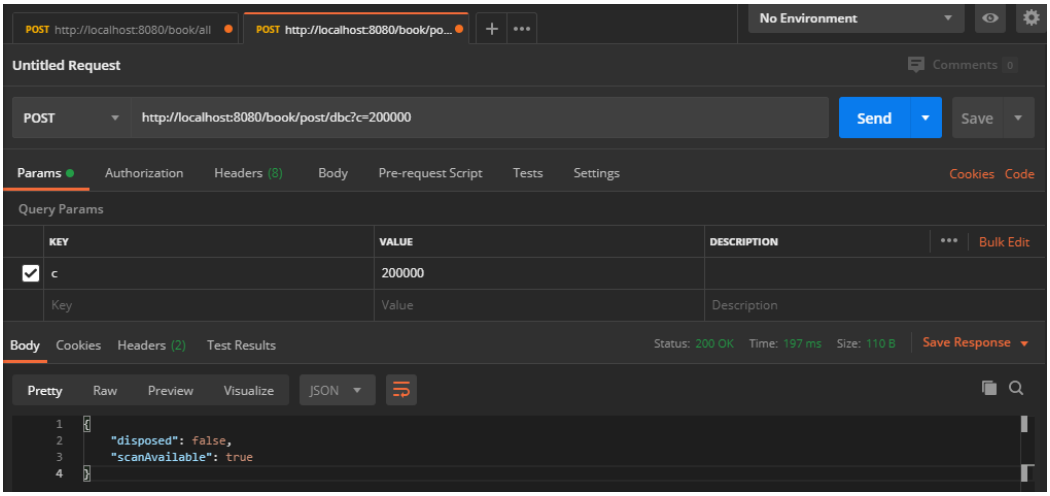

Figure 6.16: Screenshot of Postman after failed test.

Figure 6.17 shows the VisualVM output of the failed tests.

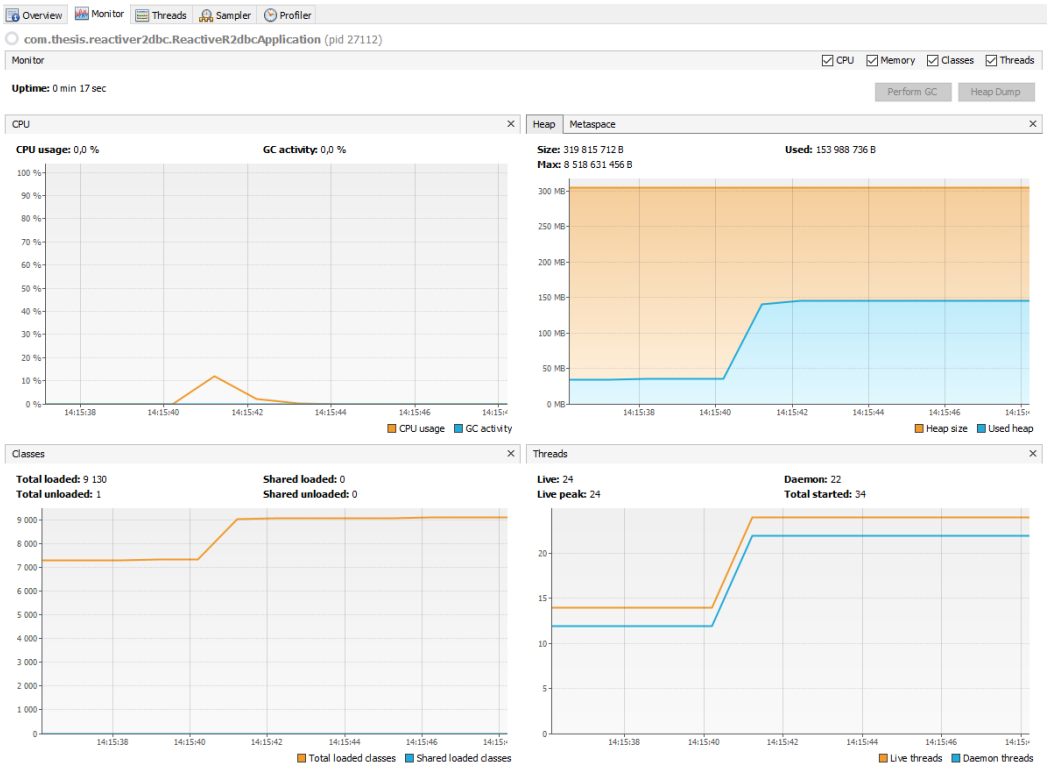

Figure 6.17: Screenshot of VisualVM after failed test.

SELECT

As stated after the first set of insert tests all tests will only be run with the reactive transactions enabled.

Since the SELECT test consists of just a single select statement. And a batch is a collection of several statements no select test was conducted using the *Batch* class.

The SELECT test with 1 000 000 values using *repository* and not streaming the reply back to the sender. Figure 6.18 is a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

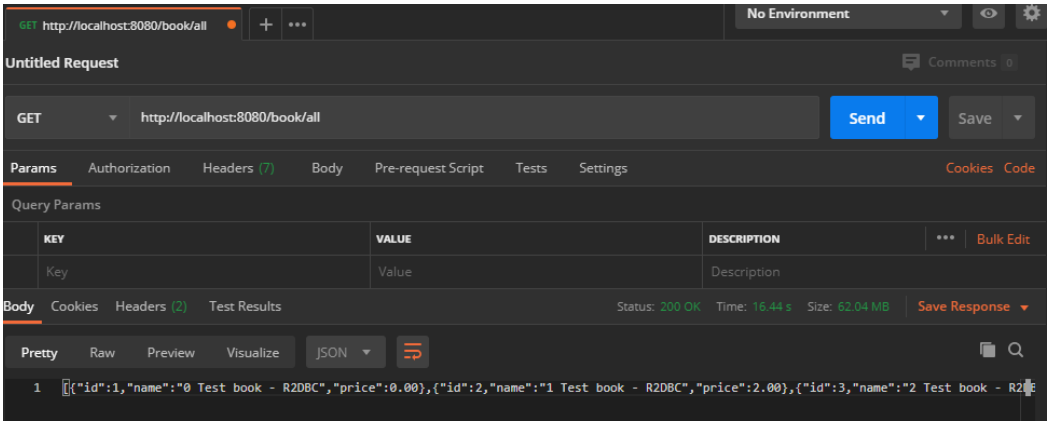

Figure 6.18: Screenshot of Postman after non-streaming select with 1 000 000 values.

Figure 6.19 shows more details on where the time the request took was spent.

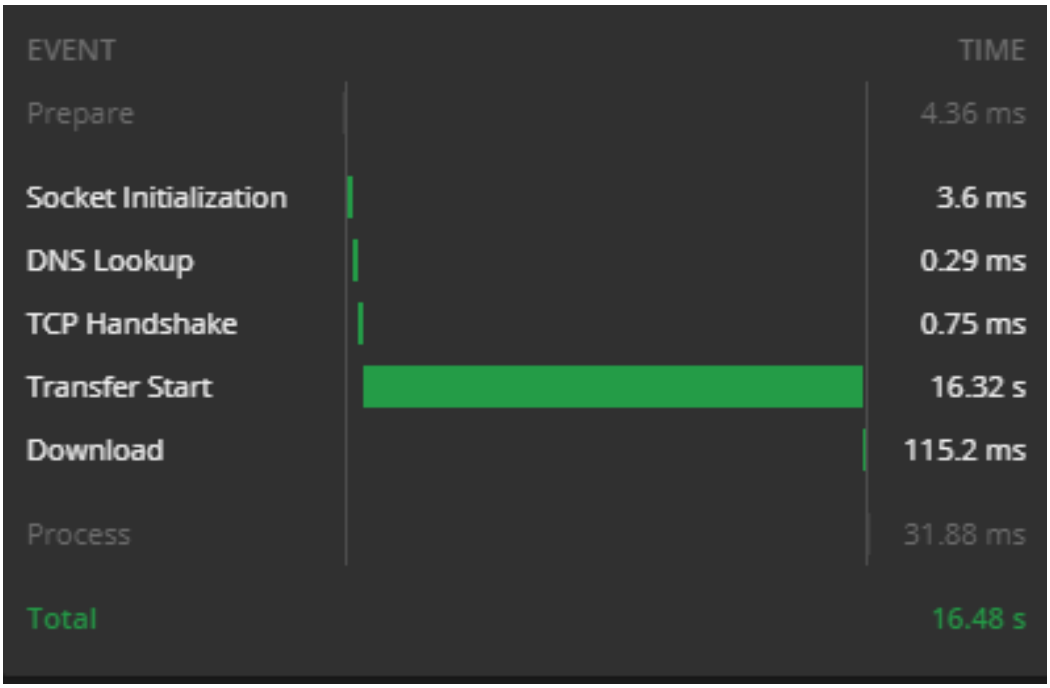

Figure 6.19: Screenshot of time details from Postman after the non-streaming select.

Figure 6.20 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when an SELECT with 1 000 000

instances of the type *Book* in the database with a *repository*. It shows the process uptime, CPU usage, RAM usage, loaded classes and threads.

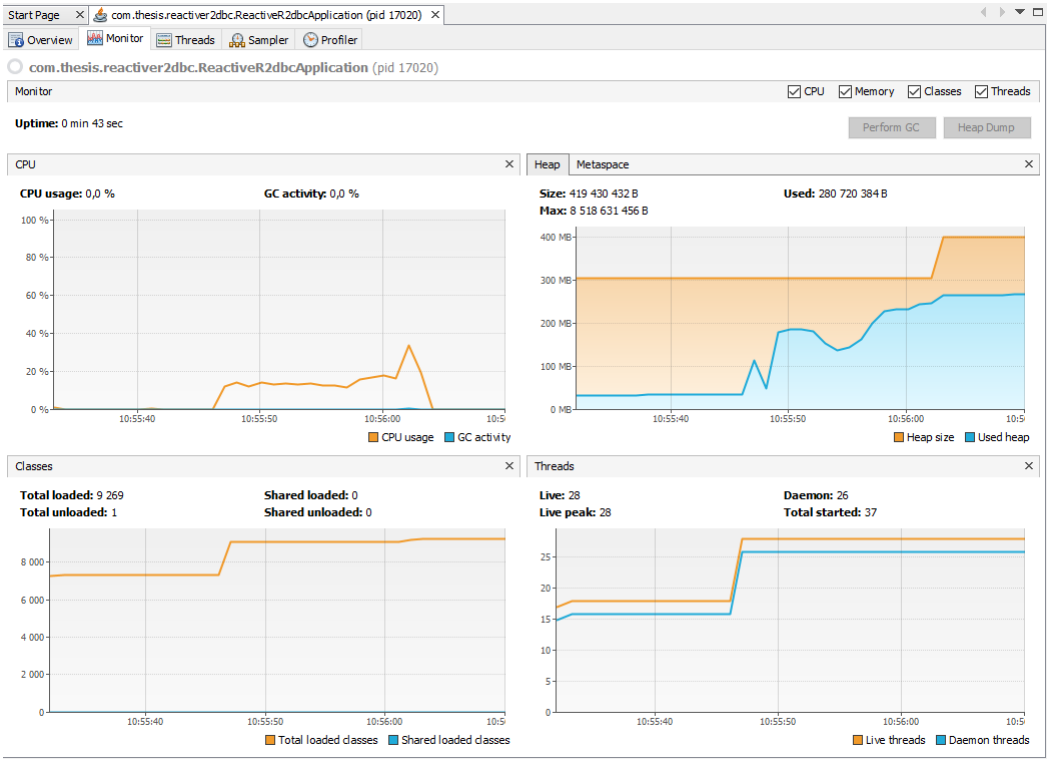

Figure 6.20: Screenshot of VirutalVM after non-streaming select with 1 000 000 values.

The SELECT test with 1 000 000 values using *repository* and streaming the reply back to the sender. Figure 6.21 is a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

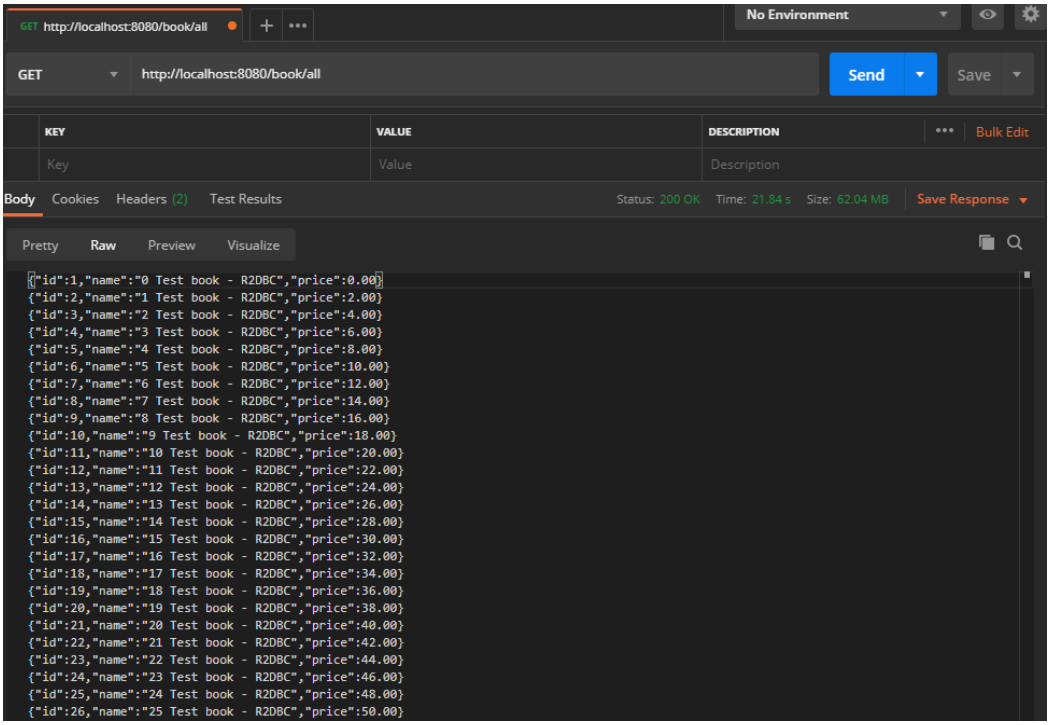

Figure 6.21: Screenshot of Postman after streaming select with 1 000 000 values.

Figure 6.22 shows a more details on where the time the request took were spent.

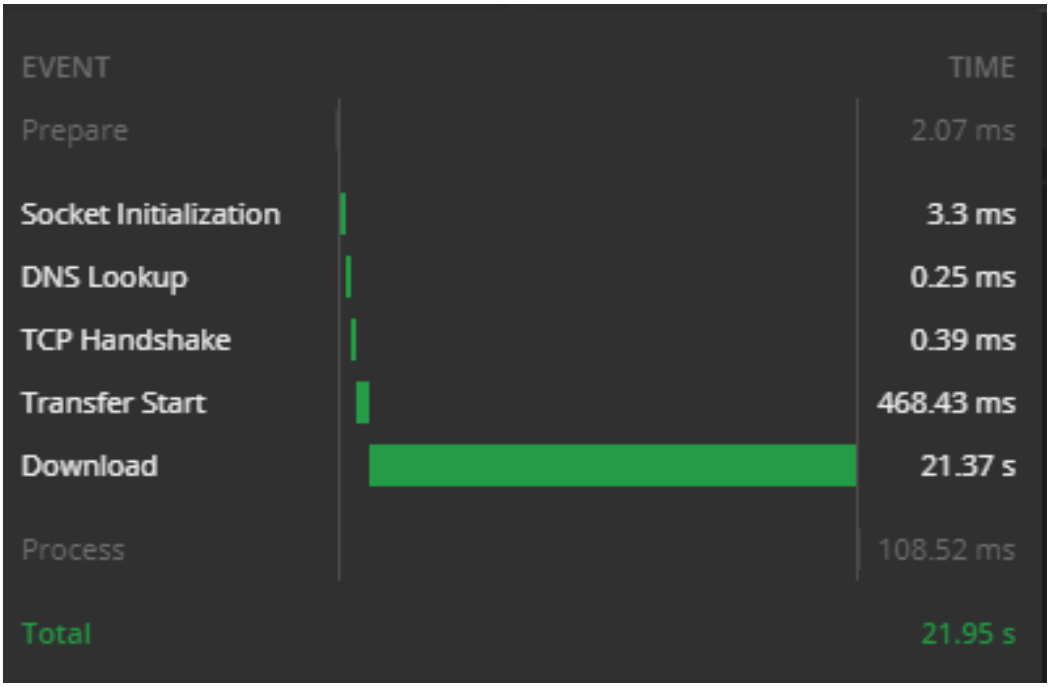

Figure 6.22: Screenshot of time details from Postman after the streaming select.

Figure 6.23 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when a SELECT with 1 000 000 instances of the type *Book* in the database was returned as a stream with a *repository*. It shows the process uptime, CPU usage, RAM usage, loaded classes and threads.

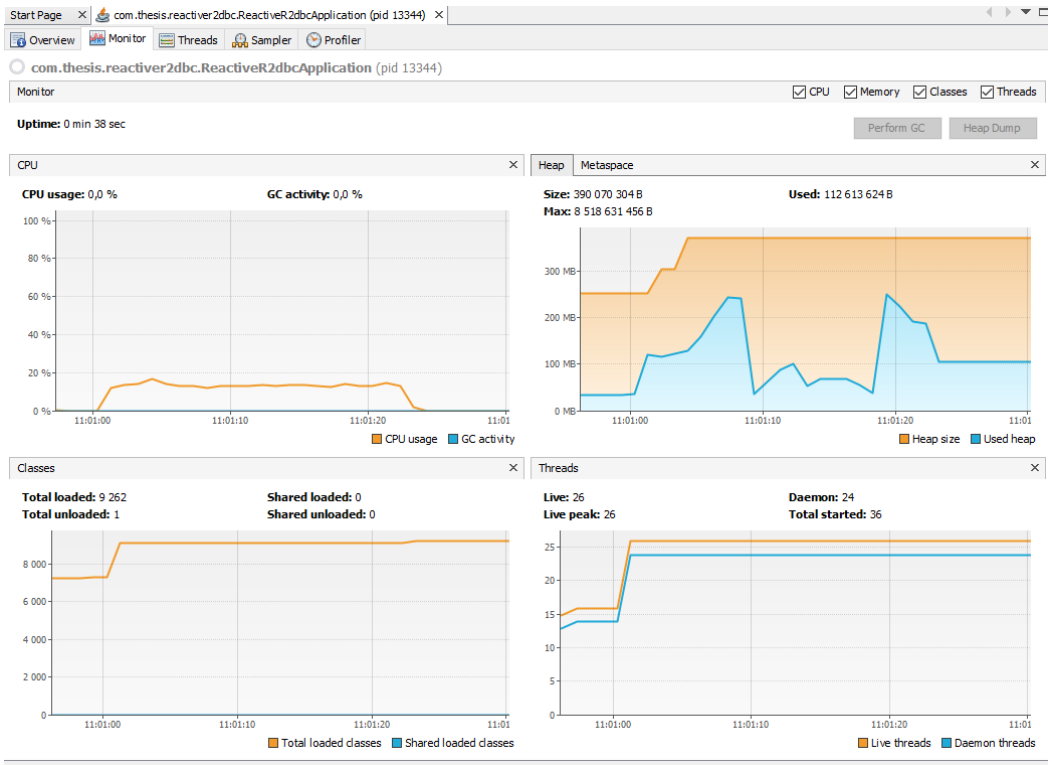

Figure 6.23: Screenshot of VirutalVM after non-streaming select with 1 000 000 values.

The SELECT test with 1 000 000 values using *DatabaseClient* and not streaming the reply back to the sender. Figure 6.24 is a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

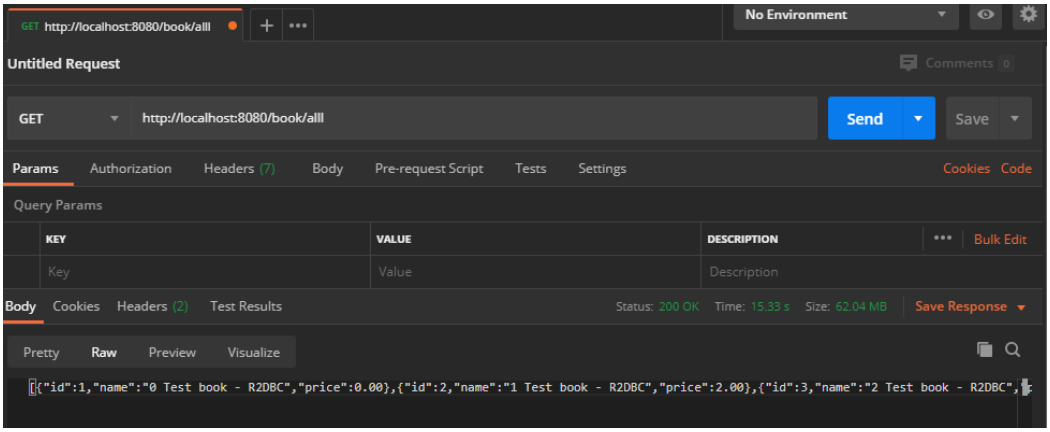

Figure 6.24: Screenshot of Postman after non-streaming select with 1 000 000 values.

Figure 6.25 shows a more details on where the time the request took were spent.

Figure 6.25: Screenshot of time details from Postman after the non-streaming select.

Figure 6.26 is a screenshot of *VirtualVM* showing the hardware metrics

for the program's Java process when an SELECT with 1 000 000 instances of the type *Book* in the database with a *DatabaseClient*. It shows the process uptime, CPU usage, RAM usage, loaded classes and threads.

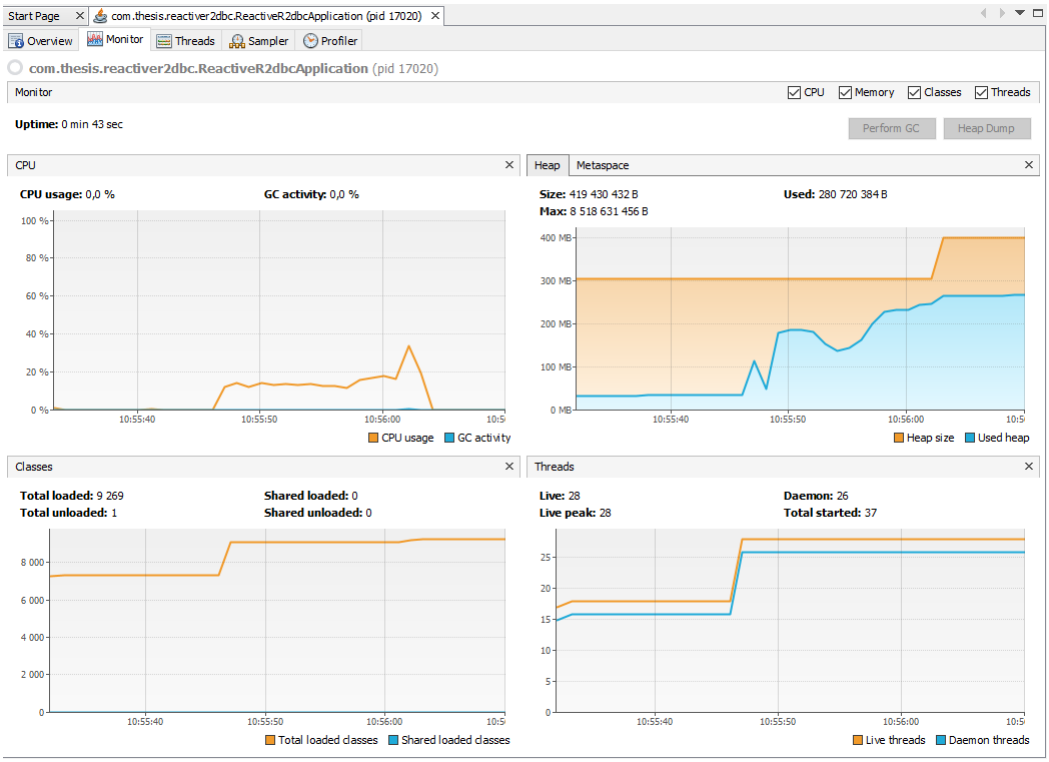

Figure 6.26: Screenshot of VirutalVM after non-streaming select with 1 000 000 values.

The SELECT test with 1 000 000 values using *DatabaseClient* and streaming the reply back to the sender. Figure 6.27 is a screenshot of *Postman* that shows the type of request and to what address it is sent, the response from the program and the time it took to get a response on the request.

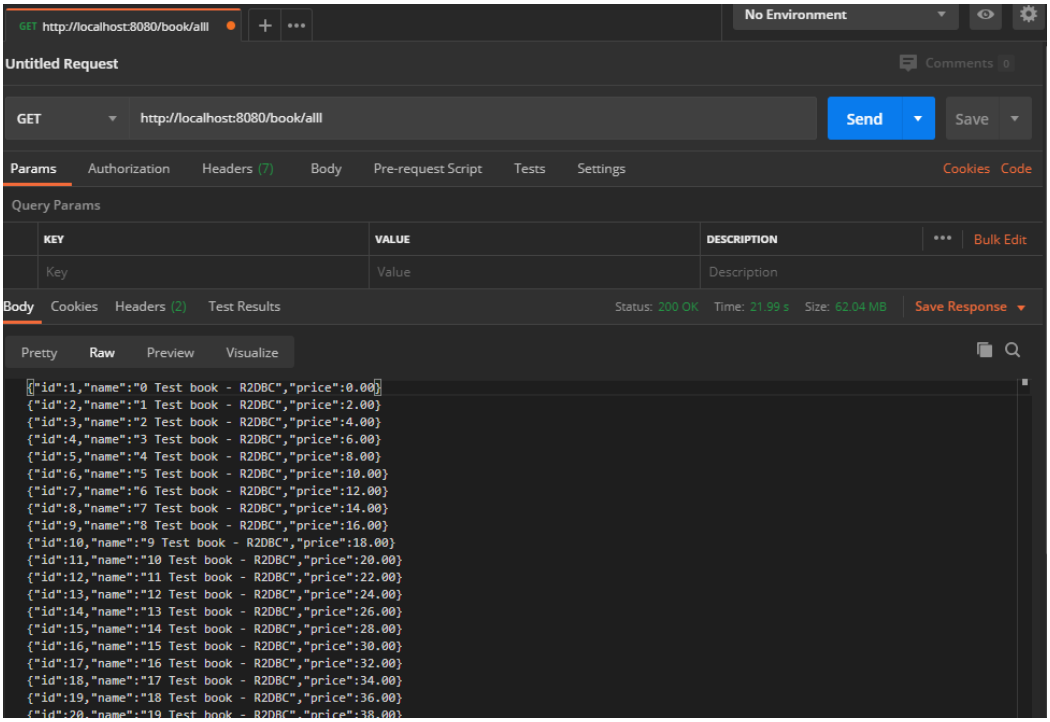

Figure 6.27: Screenshot of Postman after streaming select with 1 000 000 values.

Figure 6.28 shows a more details on where the time the request took were spent.

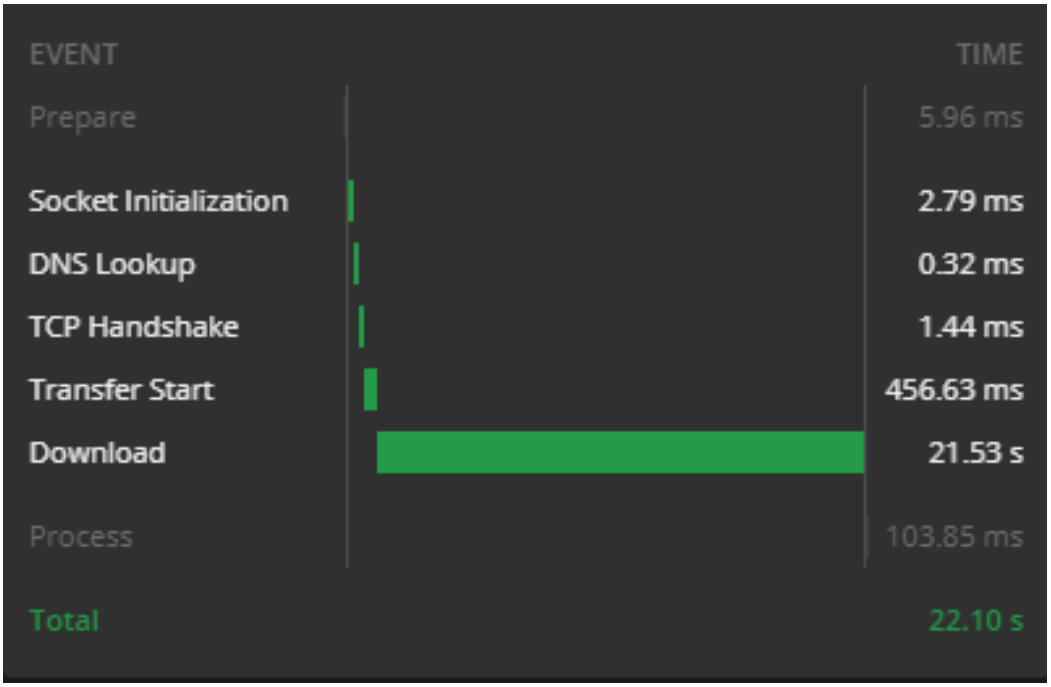

Figure 6.28: Screenshot of time details from Postman after the streaming select.

Figure 6.29 is a screenshot of *VirtualVM* showing the hardware metrics for the program's Java process when an SELECT with 1 000 000 instances of the type *Book* in the database was returned as a stream with a *DatabaseClient*. It shows the process uptime, CPU usage, RAM usage, loaded classes and threads.

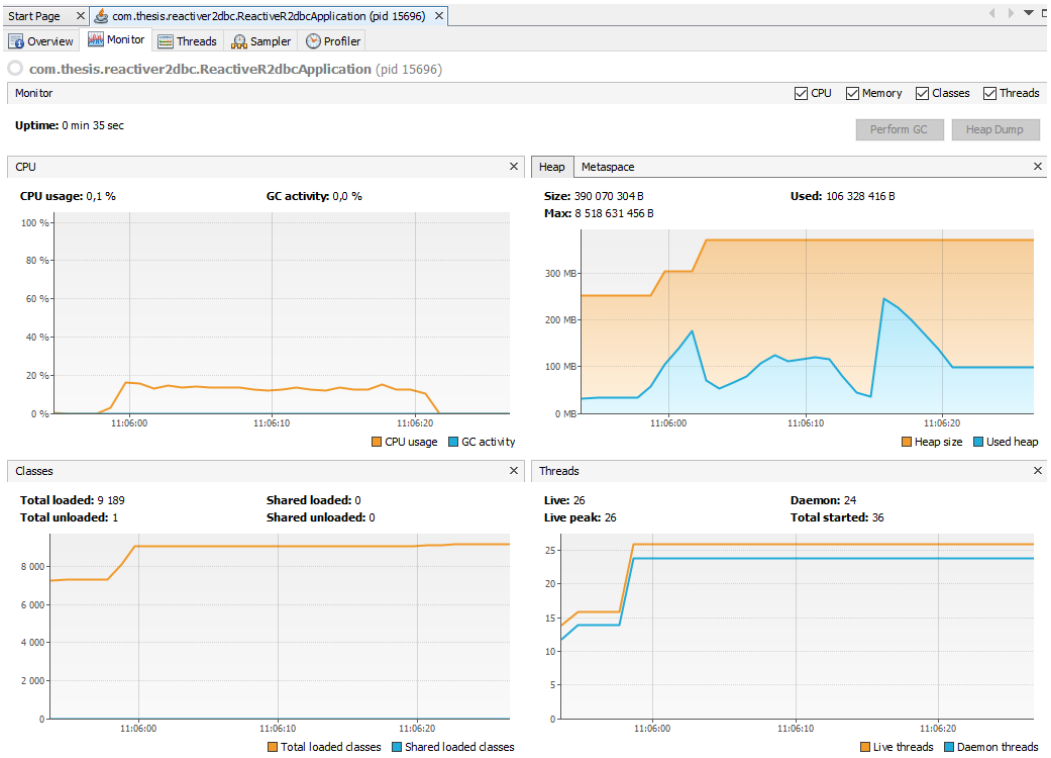

Figure 6.29: Screenshot of VirutalVM after non-streaming select with 1 000 000 values.

A set of tests was run with heap space limited to 256 MB. These tests could be completed for all select versions of the *R2DBC* program. Figure 6.30 is a screenshot of *VisualVM* after a completed non-streaming select using a *repository*.

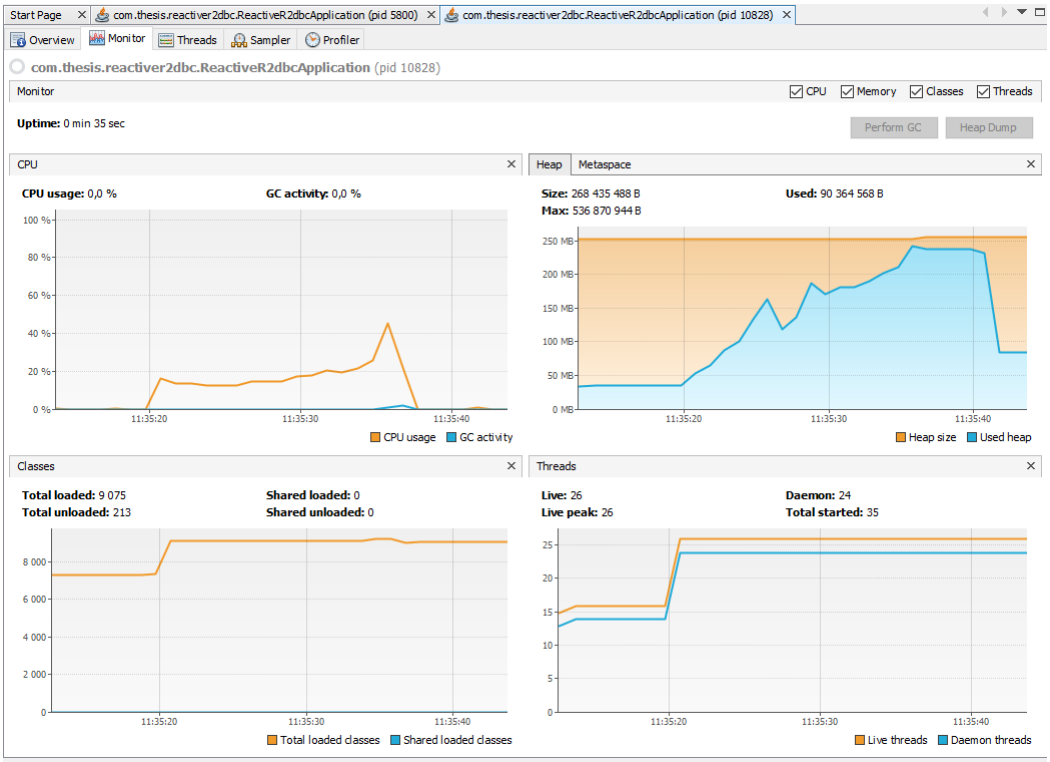

Figure 6.30: Screenshot of VisualVM after sending select command with 1 000 000 values in to the table and limited heap memory.

6.3 R2DBC program - BLOB handling

Tests to see if BLOBs could be handled without loading the entire thing into memory consisted of tried implementation of this function and monitoring of the memory usage of the application with *ViusalVM*. To test this two BLOBs of three different sizes where added to a local database on the HP laptop. Since the test was if it was possible to work with a BLOB without loading it entirely into the memory the fact that the database was run locally on the laptop the tests were run from does not matter. The three sizes of the BLOBs where 12.5 kB, 100 kB and 316306 kB (316 MB). According to the documentation of the *MySQL* implementation of *R2DBC*[22] BLOBs map to one of *ByteBuffer*, *Blob* or *byte*[]

Figure 6.31 shows the layout of the table where *id* is an auto generated number and the column *test* is of the type *LONGBLOB*.

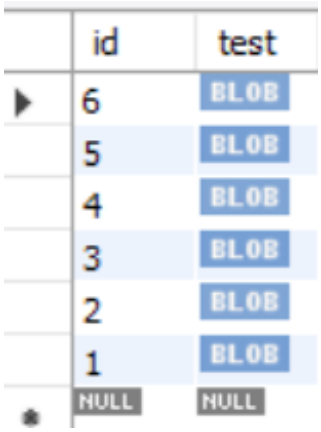

Figure 6.31: Screenshot of the layout in the table containing BLOBs.

Tests of trying to use *Spring repositories* for the table failed. Attempts to get the BLOBs as a Flux of a type like *byte*[] resulted in the entire BLOB being loaded into memory before it could be handled. And attempts to get them as a Flux of Flux of a class failed. The available heap memory was lowered to check if it could handle the BLOB without loading the entire thing into memory. But the program crashed due to it running out of memory.

7. Conclusions

Firstly I implemented programs for performance testing in accordance with what is stated in section 5.2 and what is stated in section 6.3 which resulted in two programs, one that uses *JDBC* and servlet stack and one that uses *R2DBC* and reactive stack, the features of these programs are described in section 6.1. The reason for the choices of communication in the *JDBC* program is that they were the first ways I encountered when looking for ways of communicating with a database.

In addition to the database communication ways for a reactive program mentioned in this thesis it is also possible to send SQL statements by using the *Connection* class which is just the connection to the database. And In addition to the ways mentioned for the *JDBC* program there are several other ways.

Secondly, I solved the goal *Test the following capabilities of the programs when handling large amounts of data:* and the subgoals: *CPU consumption of the programs.*, *Memory usage of the programs.* and *Measure the execution time.* I did so by inserting and selecting values from the database.

Due to the fact that one of the main points of reactive programming is that the program should never ask or send more then the program can handle. Both the inserts with the *DatabaseClient* and the *Batch* class were considered failures since they would require me to manage the batch size on my own or sending one statement at a time. Those were mostly included in the thesis to illustrate that there are other ways of communicating with a database with *R2DBC* other then *Spring* repositories. But the repositories are the main focus of the select and insert tests of this thesis. So for the insert tests the only result discussed further is the *repository* result from the *R2DBC* and the *JDBC* programs result. From the select tests the *repository* and *DatabaseClient* result from the *R2DBC* program and the result from the *JDBC* program is discussed.

The results from the *R2DBC* without the reactive transactions are not mentioned since they were so much slower than all the rest, only inserting around 90 000 in one hour and eight minutes.

The insert tests show that both the *JDBC* and the *R2DBC* programs *repository* tests used about the same amount om memory from a bit over 50 MB up to around 250 MB. But the CPU consumption is noticeably higher on the *R2DBC* program, Still not high but since it reached a maximum of 10% in the test with reactive transactions. The execution time is significantly shorter for the *R2DBC* program.

It is possible to make the *JDBC* program faster by tweaking things like the batch size and perhaps using another way of communication with the database, but a better batch size is something that needs to be set separately for each program by trial and error. This means that *R2DBC repositories* provides a much faster way of inserting values ”out of the box”.

The select tests show that the *JDBC* program was faster then the *R2DBC* program but that the *JDBC* program runs the risk of crashing if it has too little memory, which the *R2DBC* does not.

Here I would like to point out that the fetch size of *JDBC* program is like the batch size in the insert tests and tweaking of this would speed it up. But a to large fetch size or batch size runs the risk of crashing the database or the program communicating with the database. *R2DBC* also gives you this by default in theory and it would seem like it does it in reality as well since when it worked with less memory it handled it well. It only asks for the amount of data it can handle.

I would like to further point out that my programs have been constructed in a very generic way and tested and that optimization would speed things up. And that you get some of that from free with the *R2DBC repositories* that is part of why the inserts it is so much faster the *JDBC* program.

Thirdly I solved the goal *Test if it is possible to handle large BLOBs in a reactive program.* I did so by trying to implement BLOB support into the *R2DBC* program, see section 6.1.2. In my reactive program it is currently not possible to handle large BLOBs in a reactive program using *R2DBC* since it requires a program to load the entire BLOB into the memory when getting it from the database, see section 6.3. This leads to problems if the BLOBs in the database is several gigabytes in size. I have only tried with the *MySQL* driver implementation of

R2DBC since it was the only driver implementation at the time of this thesis that claimed to have *Reactive LOB types* but what that means is not explained further. From this I drew the conclusion that it is not possible to handle large BLOBs in a reactive program that uses *R2DBC* to communicate with a SQL database.

If in the future any *R2DBC* driver implementation allows each BLOB in a column that stores BLOBs to be fetched in chunks for example as a *Flux* of *byte*[]. So the final returned data would have the type *Flux*<*Flux*<*byte*[]> > when asking for the entire BLOB column where the inner *Flux* represents each BLOB in the column and the outer *Flux* represents the column. It might be possible to use *R2DBC*

Fourthly I solved the goal *Evaluate and present the result from the second, third goal and potential other observations made during the tests and draw conclusions.*

To summarize and answer the overall goal, the results in chapter 6 shows that benefits seem to be that the *R2DBC* seem to give the user a good and fast solution "out of the box" without the need to optimize things like how much to send or fetch through trial and error since it does that for you. In other words you get a lot for free since it does the best with the hardware you have got. The drawbacks are that it, seems to have a slower select, is not currently possible to handle BLOBs in a good enough way as previously explained in this section and that *R2DBC* is a very new technology and have not even reached version 1 yet and have no current driver that has implemented the whole specification. So it is a bit untested and needs more testing before it can safely be used in products.

Lastly to answer the question asked by the assignment supplier *Easit* about the benefits of switching to a reactive stack program. I would say that they are not big enough to warrant a switch to a reactive program from a servlet program. Because even though a hardware resource point of view the thesis shows that the *R2DBC* might give faster inserts in my case I assume that they have a better and more developed *JDBC* program and handles lower memory better then *JDBC* program. Since the one tested here was very simple. Some drawbacks are the fact that *BLOBs* is currently loaded into memory in its entirety before it can be used means that BLOBs can not be handled in a good enough way

currently with a reactive program using *R2DBC*. The fact that I during the process encountered a steep learning curve when trying to learn and use reactive *Java* means that this is probably the case for most Java developers since reactive code differs quite a bit from ”regular” code and if it were to be used extensive rewrites of current code would probably be needed to make it reactive.

7.1 Future work

There are several different possibilities for future work. One is a bigger and more detailed comparison between a non-reactive program and a reactive program. Even here there are different possibilities some of these are compare more ways of communicating with a database, use a different data structure, for example, large ones but fewer for tests since this one was very small 3 columns and a lot of *books*, more extensive comparison not just the execution time of selects and inserts and include more *R2DBC* driver implementations in the tests, not just the *MYSQL* implementation.

Another direction to go is a more general comparison between reactive and non-reactive Java since this thesis is focused on the difference between *JDBC* and *R2DBC*, in other words, database communication.

One direction to take future work is to evaluate and try to map what other code changes are needed to switch from servlet to reactive stack. This would be very dependant on the software but some general points and estimations could be produced if the study is large enough.

Another thing that can be seen as future work is to redo the comparison when the drivers have been developed further. For example when a driver has implemented the entire version 1.0 *R2DBC* specification.

7.2 Ethical considerations

Since reactive programming and asynchronous database communication are not new concepts in themself, just in Java. I can not see any new ethical considerations, just the ones that are always present when developing software. For example the fact that the data should always be safe and not accessible to people that should not have access to it is one part of it. But since *R2DBC* and reactive *Java* only change how the data sent is handled (the use of *Mono* and *Flux*) and not the underlying

transfer protocols it does not add any new things from this perspective. It is the same as all the other database drivers and as previously stated does not create any new ethical considerations, but makes it so that if there are any special ethical considerations needed when working with reactive programming and asynchronous database communication needs to be taken in to account when using Java.

References

[1] TIOBE, "Tiobe index for february 2020." https://www.tiobe.com/tiobe-index/. Accessed: Feb 2020.

[2] TIOBE, "We measure your software code quality." https://www.tiobe.com/company/about/. Accessed: May 2020.

[3] S. Tibken, "Ces 2019: Moore's law is dead, says nvidia's ceo." https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces/ /-2019/. Accessed: apr 2020.

[4] J. Shalf, "Hpc interconnects at the end of moore's law," in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*, pp. 1–3, IEEE, 2019.

[5] B. L. Tracy Lee, "Reactive programming is not a trend: Why the time to adopt is now." https://news.thisdot.co/reactive-programming-is-not-a-trend-why-the-time-to-adopt/ /-is-now-b3954ea44aa3. Accessed: Apr 2020.

[6] T. Nurkiewicz and B. Christensen, *Reactive Programming with RxJava: Creating Asynchronous, Event-based Applications*. " O'Reilly Media, Inc.", 2016.

[7] F. Gutierrez, *Pro Spring Boot*. Springer, 2016.

[8] J. Deacon, "Model-view-controller (mvc) architecture," *Online*][*Citado em: 10 de março de 2006.*] *http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[9] R. K. Jonas Bonér, Dave Farley, M. T. W. the help, and refinement of many members in the community., "The reactive manifesto." https://reactivemanifesto.org/.

[10] "Reactive streams." https://www.reactive-streams.org/.

[11] J. R. Groff, P. N. Weinberg, and A. J. Oppel, *SQL: the complete reference*, vol. 2. McGraw-Hill/Osborne, 2002.

[12] Y. Bai, *JDBC API and JDBC Drivers*, pp. 89–111. 2011.

[13] Ben Hale, Mark Paluch, Greg Turnquist, Jay Bryant, "R2dbc - reactive relational database connectivity." https://r2dbc.io/spec/0.8.1.RELEASE/spec/html/. Version 0.8.1.RELEASE, 2020-02-04.

[14] R. Saxena, "Springboot 2 performance — servlet stack vs webflux reactive stack." https://medium.com/@the.raj.saxena/springboot-2-performance-servlet-stack-vs-webflux\-reactive-stack-528ad5e9dadc.

[15] Reactor, "reactive-streams-commons." https://github.com/reactor/reactive-streams-commons.

[16] Reactor, "Reactor." https://github.com/reactor/reactor.

[17] Pivotal, "Web on reactive stack." https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html.

[18] ReactiveX, "Rxjava: Reactive extensions for the jvm." https://github.com/ReactiveX/RxJava.

[19] ReactiveX, "Rxjava 1.0." https://github.com/ReactiveX/RxJava/milestone/2?closed=1.

[20] R2DBC, "Reactive relational database connectivity postgresql implementation." https://github.com/r2dbc/r2dbc-postgresql. Accessed: feb 2020.

[21] R2DBC, "Reactive relational database connectivity microsoft sql server implementation." https://github.com/r2dbc/r2dbc-mssql. Accessed: feb 2020.

[22] Mirro Mutth, "Reactive relational database connectivity mysql implementation." https://github.com/mirromutth/r2dbc-mysql. Accessed: feb 2020.

[23] VisualVM, "Download." https://visualvm.github.io/download.html.