JÖNKÖPING UNIVERSITY
School of Engineering

GraphQL query performance comparison using MySQL and MongoDB

By conducting Experiments with and without a DataLoader

Main Subject area: *Computer Engineering*
Author: *Didrik Nordström, Marcus Vilhelmsson*
Supervisor: *Jasmin Jakupovic*
JÖNKÖPING 2022 September

This final thesis has been carried out at the School of Engineering at Jönköping University within Computer Engineering. The authors are responsible for the presented opinions, conclusions, and results.

Examiner:	Anders Adlemo
Supervisor:	Jasmin Jakupovic
Scope:	15 hp (first-cycle education)
Date:	2022-09-26

Abstract

GraphQL is a query language rising in popularity, causing many transitions from traditional API endpoints to a GraphQL solution. Reflecting upon the positives and the flaws to using GraphQL, a DataLoader for batching queries sent to databases is sold as a solution to the infamous N+1 problem. Experiments were conducted to test how GraphQL response time, with and without DataLoader, changes when paired with MySQL and MongoDB. Along with the experiments, a Literature Review was conducted reflecting over the databases structural differences that could affect the response time for GraphQL. Results suggest that no major differences were to be found, and the explanation for the minor differences could rather be because of the disparity in query optimization instead of architectural differences for MySQL and MongoDB.

Keywords DataLoader, GraphQL, MongoDB, MySQL, Query, Query Performance

Table of Contents

1 Introduction

The following subsections will describe the *problem statement and motivation* followed by *purpose and research questions*, *scope and limitations* and the *disposition* of this report. It will give you, the reader, an insight into why there is a problem to begin with, and the reason the authors wanted to investigate the issue.

1.1 Problem statement and motivation

Software systems are complex structures and several choices between software (language, architecture, packages etc.) and hardware must be made when designed. It is common for companies to transition from one technology to another if it suits them better at that time. Verendus System AB was facing such a choice, wanting to transition their backend external facing API's (Application Programming Interface) from REST to GraphQL. Verendus is a company providing Enterprise Resource Planning (ERP) solutions to recreational vehicle (RV) dealers. APIs are used to fetch or manipulate data on a server. REST works by providing endpoints for the user to interact with while GraphQL binds data to an object and allows specific data to be fetched or manipulated through queries.

Each software is unique and comes with both advantages and disadvantages. Preceding study has also been done regarding the migration from REST to GraphQL which highlighted some fundamental differences between the different technologies. The study shows an adaptation of GraphQL services may in some cases conflict with the design principles in the original architecture (Vogel et al. 2018). This piqued the authors of this report's interest into what difference the structure of the backend of a system would make with a GraphQL endpoint.

Using GraphQL with MySQL and MongoDB has been performance tested in previous research, displaying results of MongoDB accompanied by GraphQL performing worse than MySQL when one row is requested compared to 100 rows in a single request (Erlandsson & Remes, 2020). We believe that this does not tell the whole truth about data-fetching with GraphQL, since nested objects need to have a DataLoader to be efficient in its fetching for multiple data (Rinquin, 2017). Under-fetching issues with the N+1 problem (requests sent to the server equals number of objects retrieved (n), plus one extra request) is caused by having divided requests when retrieving data. This often leads to unnecessary requests to the data source (Lundqvist, 2020).

Because of the N+1 problem, the authors wanted to dive deep into GraphQL and compare the query performance between using a SQL database (MySQL) and a NoSQL database (MongoDB) and investigate what can cause a potential difference in performance. The authors wanted to investigate if GraphQL favors one database type over the other in terms of response time performance. Meaning, does the performance difference between the two databases paired with GraphQL solely depend on the underlying database type or does GraphQL performance depend on other factors like how the data is fetched?

1.2 Purpose and research questions

GraphQL is currently gaining in popularity according to a study made by Cloud Elements (Geene et al., 2022) in 2021 where a total of 75% of their respondents believe that GraphQL will become the predominant API style for their business in the future. Comparing this to their 2020 study, only 40% said they believed so.

Popularity is one thing, but little do we currently know about the importance of how the underlying databases could affect the query performance while using this new end-point technology. As the performance measurements made by Erlandsson & Remes (2020) display it is a clear difference when fetching more than one resource from a data-source, the authors would question themselves why and what is the cause of this. Therefore, the first research question is as follows.

1. How do query retrieval times differ using GraphQL with MySQL compared to GraphQL with MongoDB?

Our first thoughts were that Erlandsson and Remes in their research used Collections to separate data in their MongoDB database structure which then causes the need of multiple queries of retrieval, slowing down the performance when fetching (MongoDB Model Relationships, n.d.). Hence the reason for our second question, where we will examine why any differences between these solutions could occur.

2. What impacts the potential query performance differences or similarities between these different database solutions?

1.3 Scope and limitations

1.3.1 The scope of this study

The design behind the databases is made to represent both a fair comparison between MySQL and MongoDB but also show a real presentation of real-world systems. But as is the nature of experiments, a case must be made which may or may not be applicable in other use cases. Therefore, different collections for each object are made in MongoDB. While this may not be the most optimized structure for a real-world use case, it is still a valid structure that enables a fair comparison between these databases. We choose to only evaluate the response time for each system locally, without the added variable of internet speeds. While this ensures that the results are comparable to each other, it cannot ensure that it is the case whilst running systems over the internet with variable internet speeds.

1.3.2 The limitations of this study

The study does not contain comparisons between other database management systems paired with a GraphQL API than the two previously mentioned. Performance in terms of latency is only measured for Read operations, and other operations like Update and Insert etc. are not considered.

There are other factors that can impact the performance of these systems that are out of our control. The experiments are conducted on a PC running Windows, and factors like hardware utilization done by the operating system itself are hard to limit. To avoid these

factors tampering with our results we chose a large test sample size to find a trend of performance across all tests.

Any differences between these databases will not be attempted to be solved in such an extent to bring a solution due to the restrictions of time. The purpose is to highlight the issues and what causes them. We hope instead that this research can be of use in future research to solve this potential issue.

1.4 Disposition

The paper will firstly present chapter *2 Theoretical framework*, where key terms are explained along with other information that may be needed to fully understand this study. Chapter 3 *Method* gives a brief overview of how the data is collected and brings up key areas such as validity and considerations for our research. Chapter 4 *Implementation* describes steps taken to create the experiments and in chapter *5 Results* are presented which are discussed in chapter *6 Discussion.* The report is summarized and ends with our own conclusions in chapter *7 Conclusions and further research.*

2 Theoretical framework

The theoretical framework will explain relevant terms frequently used throughout this report and give a brief explanation of their usage. We recommend going through each term in the intended order to ensure the reader gets a complete understanding of how each term, technology or standard mentioned interacts with each other.

2.1 HTTP

Hyper Text Transfer Protocol (HTTP) is the protocol behind the World Wide Web. Transactions in HTTP works by requests and responses. Each request and response contain a header with relevant information such as Media types, which specifies in which format the data is transferred in (Wong, 2000). The media type used in this report case study is JSON.

2.2 JSON

JavaScript Object Notation (JSON, n.d.) is a lightweight data interchange format that is completely language independent but has similar annotations found in other languages from the C-family. It was designed with human readability in mind, and it is also easy for computers to parse and generate. JSON has five different datatypes, *string (double quotes*), *number*, *Boolean* (*true* or *false*), *object (curly brackets)* or an *array (square brackets)*, where the last two structures can be nested. The data interchange format is best used with JavaScript based systems. (JSON, n.d.)

In our experiments JSON is the preferred data interchange format because of its lightweight easy to read format (See figure 1).

```
{
  "firstname": "Alice",
  "lastname": "Andersson",
  "age": "42"
}
```

Figure 1: Example of JSON object describing a person

2.3 API

An Application Programming Interface (API) is used to connect software systems over the internet while hiding implementation details. They enable loose coupling of systems across all platforms and several *Frontend* systems like websites or mobile applications can all make use of the same API's. An API solution commonly consists of two types of components, one exposing and another consuming. The frontend acts as a client in an API solution and consumes the API, whereas the server-side, also known as the backend, exposes the API in the architecture usually from the cloud or on premise (Biehl, 2015).

A frontend program often displays information to the user that they interact with and serves as the gateway between the user (client) and the backend (server). Backend programs react to the requests and receives, stores, and responds with data. It enables the two different systems to communicate over the internet and can also mean several

frontend systems are connected to one backend. APIs enable both internal and external systems to communicate for better workflow while still offering a layer of security. Requests made with an API are based on URL's (Unified Resource Locater) where each resource accessible by the API is identified with a URL (IBM Cloud Education, 2020). In figure 2 below the requests and responses are shown in a data flow from the clients to the databases.

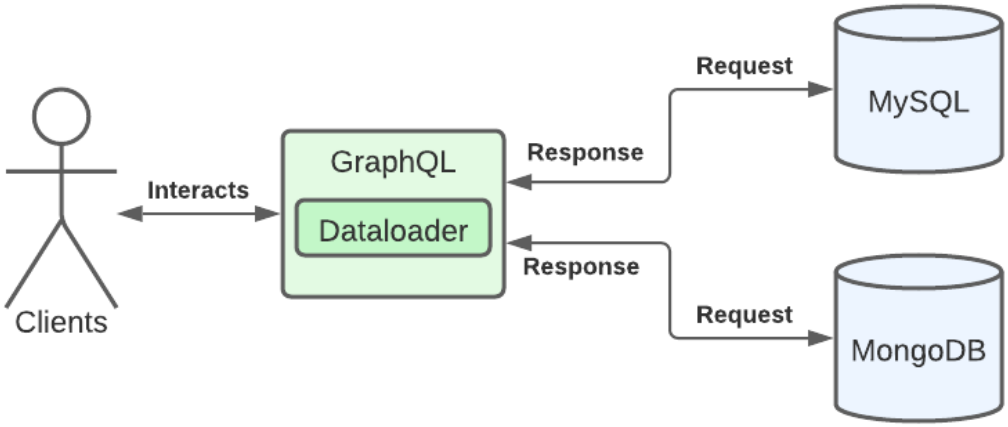

Figure 2: Data flow from Client to Databases

2.3.1 GraphQL

GraphQL is a query language for an API, meaning to access data a query must be sent specifying the fields which the client wishes to receive. This is different from the REST-architecture, as it is based on endpoints which deliver data determined by the server mounted at that endpoint. GraphQL enables selective data fetching of resources which leads to smaller packet size on return. It was founded by Facebook in 2012 and became open source in 2015. (GraphQL Foundation, n.d.)

GraphQL and its relevance in this comes down to Verendus wanting to adapt to this technology because of its previously mentioned features.

GraphQL Queries

GraphQL queries depend on whether the response fields will reflect upon the fields that were requested. This to assure that you always get back what you expect. This can also be combined with the option to pass an argument with the query. (GraphQL Foundation, n.d.).

In the example on the next page (figure 3) a query with an *id* parameter is displayed along with the response from the request.

Request:

```
{
  Library(id: 1) {
    id,
    name,
    street
  }
}
```

Response:

```
{
  "Library": {
    "id": 1
    "name": "foo"
    "street": "bar street 1"
  }
}
```

Figure 3: Request and response from application using GraphQL

N+1 problem

The so called “N+1” problem is exclusive to GraphQL and is caused by how it manages data fetching with nested objects. If we were to fetch a nested object like in figure 3, with an API implemented using REST, it would result in two queries hitting the database but only one sent from the client. The first query collects all the entries of the first object “author”, and the second query collects all the entries for the nested object (books of that author). GraphQL manages this scenario by causing the number of queries to be sent to the database to be 1 + N, where N is the number of objects returned, hence the name “N+1 problem”. This will cause issues when the requests are in substantial number of objects, as it scales linearly. The explanation for this behavior is how resolver function only knows of its parent object, meaning the second resolver function only knows to look for the books related to its parents' author. In the example below it would result in four calls to the database, one call to get the information needed from the resolver function, one call for the author and two calls for his books. There is, however, a solution package called DataLoader. (Cronin, 2019)

```
{
  "name": "Library of FooBar town",
  "street": "Foobar street",
  "books": [{
    "title": "Alice in FooBarland",
    "genre": "horror"
  }, {
    "title": "Bob's Story",
    "genre": "drama"
  }]
}
```

Figure 4: JSON object representing an author and his books

DataLoader

DataLoader is a package aimed to solve the previously named “N+1 problem” and was released in 2015. It works by batching all queries requested by the resolver functions into one large request. Fetched values can also be cached to eliminate reflecting of data

if requested again. (GraphQL, 2022) There are other solutions that work in a similar fashion, one being the Prisma ORM (Object Relational Mapping) for MySQL databases. (Prisma, n.d.)

DataLoader is of relevance to our thesis as it solves a clear disadvantage GraphQL proposes and is a recommended implementation by the developers of GraphQL.

GraphQL Bench

GraphQL bench is a simple and versatile tool that supports both HTTP and WebSocket tests to examine benchmarks and load-tests (Hazura, 2022).

To run GraphQL Bench a config file is provided with information such as:

- Query name
- Execution Strategy
- Requests Per Second
- Duration
- Connections
- Query

GraphQL Bench can be run as a Command Line Interface (CLI) application, providing an interface to see the metrics and data provided with each test run (Hazura, 2022). CLI is an interface for executing commands via text input to an operating system. This allows programs such as GraphQL Bench to run at command and enables programs to give text feedback to the user by printing warning or messages to the CLI window. (Stephenson, 1999)

2.3.2 REST

REST stands for Representational State Transfer and is an architectural style used together with API's (Application programming interfaces). REST is not a code standard, but a set of constraints used when developing your API and is built on top of the HTTP protocol. REST works by exposing different endpoints to the client that are used to access a predetermined set of data. The client can interact with the endpoints to fetch and update data on the server. One key constraint of REST is that it is stateless. This means no data is stored on the server or client between several requests, as each request is treated as its own. REST supports both the XML and the newer JSON format when transferring data. (Patni, 2017)

2.4 DBMS

Database management systems (DBMS) are large collections of software that carry out processes needed for storing, maintaining, optimizing and more for collections of data stored in files. (Ince, 2019) The general term often used is Databases, but a database can be physical records stored on paper while DBMS are entirely computerized. DBMS can be structured with tables and rows like in MySQL, key value pairs like in Redis or JSON-like objects within documents like in MongoDB. (IBM, 2010)

Schemas

Schemas is the visual representation of a database, referring to a set of rules the database consists of or a full set of objects belonging to a certain record. It is the logical configuration and a data model. (Lucidchart, n.d.)

2.4.1 MySQL

MySQL is a relational database management system (RDBMS) and is used to store data. A relational database is made up of a collection of tables that store structured data. Each table has rows of information, where each row is a single entry with attributes. MySQL uses Structured Query Language (SQL) to manipulate data. (Assaf, 2022)

2.4.2 MongoDB

MongoDB is a document database with scaling and ease of development in mind when designed (MongoDB Manual, n.d.). A document database offers several types of advantages, such as:

- An intuitive data mode that is fast and easy for developers to work with
- A flexible schema that allows for the data model to evolve as application needs change
- The ability to horizontally scale out

The listed advantages are why document databases and MongoDB are used in a variety of use cases and industries. (MongoDB Document Databases, n.d.)

NoSQL

NoSQL databases such as MongoDB are non-relational databases, meaning that data is not stored in fixed rows and columns but instead in records and documents. This results in a more flexible database structure (MongoDB Document Databases, n.d.).

Record

A record in a MongoDB database is a document, in which a data structure is composed of field and value pairs. The structure is similar to a JSON object where some objects or documents can be nested (MongoDB Manual, n.d.). See example below. (Figure 5)

Figure 5: MongoDB record (MongoDB Manual, n.d.)

Collections

A collection in MongoDB is a group of documents. Typically, collections have documents with similar contents but are not required to maintain flexibility (MongoDB Document Databases, n.d.).

2.5 Big-O Notation

The Big-O Notation is a measurement to give information about runtime complexity of algorithms and functions depending on the input n. Limitations are made depending on other functions and upper boundaries, as displayed in the example below where f is in Big-O of g and C is an arbitrary constant larger than zero (Dedov, 2020).

$$\exists N > 0 \ \wedge \ \exists C > \ 0 \colon \forall > N \colon f(n) < g(n) * C$$

$$\rightarrow f(n) = O(g(n))$$

Figure 6: Mathematical definition of Big-O Notation (Dedov, 2020)

The Big-O Notation for O(n) is of linear time, meaning that the growth is constant compared to O(log(n)) where the problem size is halved with each iteration, an example of this is binary search in a sorted list. Another example is O(1) or constant time, where the complexity does not grow no matter the input size (Dedov, 2020).

3 Method

In the following chapter, the methods of data collection and analysis, both through literature and experiments, will be discussed.

3.1 Data collection

The data collected in this thesis is split into two parts. We first collected data through a literature review with the main goal of finding research about the structural database differences, but also finding enough information about the subject. The second part of the data collection is through experiments, where test environments was set up with GraphQL and the two databases of our choice, MySQL, and MongoDB.

3.1.1 Literature Review

GraphQL suffers from known issues such as the N+1 problem, or superfluous Database calls. These problems have been studied and solutions are available but provide a hindrance layer for potential adapters of GraphQL. There are also some problems which seem to not have a direct solution, and one such problem arises when fetching too many nested fields at once. (Wieruch, 2018).

As mentioned in the Problem statement, comparison between GraphQL and other API frameworks shows that GraphQL is slower in response time. The comparison did not make use of a DataLoader to batch queries and cache results, even though it is recommended by the GraphQL Foundation (n.d.).

The results also show that MongoDB paired with GraphQL performed worse than MySQL paired with GraphQL when fetching using joins. The authors were interested in designing a test case where the use of DataLoader were present with the previously mentioned pairings of databases and GraphQL, as this would better relate to implementations in the real-world and supply more relevant results. It will also give insight into if the underlying database does any real difference in response time performance while using a DataLoader, or if the performance difference is negligible.

An API needs to be paired with a database for data to be stored. Two such databases are MySQL and MongoDB who, like REST and GraphQL, work in diverse ways. MySQL is for storing structured data with relations, while a NoSQL database like MongoDB is suited for larger datasets where relations between data is not as important. NoSQL databases have the edge in terms of performance compared to a relational database like MySQL. (Jose & Abraham, 2020). Still, the test results show that the performance difference between GraphQL paired with MySQL and GraphQL paired with MongoDB fluctuates, and there is no real winner between the scenarios presented (Erlandsson & Remes, 2020).

In this Literature Review the authors therefore investigated what could affect the query performance from the structural differences of the databases.

3.1.2 Experiments

The experiments consist of diverse types of tests conducted with GraphQL against the MongoDB database and MySQL database.

Conducting experiments to discover potential query performance differences first involved setting up two types of databases with similar set of schemas, a database schema is a visual representation of how the data is stored in the database. The schema is chosen to mimic the one used within Erlandsson & Remes experiments. And is seen in the example below (figure 7). The same structure is applied across both Database types and the related data is fetched using their respective id.

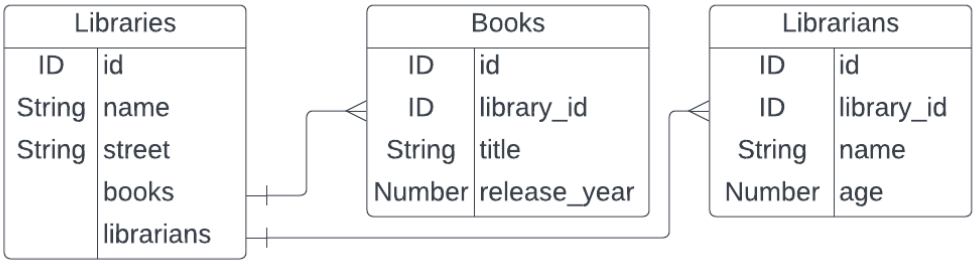

Figure 7: Database Schema Structure

The databases were then filled with data through the Faker application. Library has 10 entries made to the databases. 1000 and 20000 entries were made in separate test cases to Books and Librarians for each Library. This is to provide enough data to fetch from throughout our tests. To benchmark these tests, we used GraphQL Bench to send the requests and query each test. In all of the test cases a DataLoader is also utilized, a package aimed to solve the N+1 problem by batching queries before sending them to the server and caching the response data (GraphQL, 2022).

Erlandsson & Remes (2020) previous research and tests display that MongoDB combined with GraphQL perform worse than GraphQL with MySQL while fetching data using a single join and multiple joins. We wanted to both confirm this and see what results may vary when using a DataLoader.

The test cases have a duration period of 60 seconds and a warm-up period of 10 seconds where the results will not be recorded. The requests made per second will be 100 and 200 using 20 connections for each test. There will be 18 test cases and they will be performed in the list below accordingly. On the next page (Table 1) is a table of test cases for a better overview of the parameters changed during each test case.

Test Case 1

Get 5 entries from Library.

Test Case 2

Test Case 1 but also 1000 entries from Books per Library with a join.

Test Case 3

Test Case 2 but also 1000 entries from Librarians per Library with a join.

Test Case 4-6

Same as Test Cases 1-3 but 10 entries instead of 5 from Library.

Test Case 7-12

Same as Test Cases 1-6 but 20000 entries instead of 1000.

Test Case 13-18

Same as Test Cases 1-6 but 500 requests per second.

Table 1: Table of test cases

Test Case	Libraries	Books per Library	Librarians per Library	Requests per second	Duration
1	5	0	0	100/200	60
2	5	1000	0	100/200	60
3	5	1000	1000	100/200	60
4	10	0	0	100/200	60
5	10	1000	0	100/200	60
6	10	1000	1000	100/200	60
7	5	0	0	100/200	60
8	5	20000	0	100/200	60
9	5	20000	20000	100/200	60
10	10	0	0	100/200	60
11	10	20000	0	100/200	60
12	10	20000	20000	100/200	60
13	5	0	0	500	60
14	5	1000	0	500	60
15	5	1000	1000	500	60
16	10	0	0	500	60
17	10	1000	0	500	60
18	10	1000	1000	500	60

Specifications

The tests have been run on a computer using the Windows 11 operating system with the specifications:

- CPU: Ryzen 5 3600 3.6 Ghz
- RAM: 16GB DDR4 3200Mhz
- SSD: M.2 WD Blue SN550 (2400 MBps reading speed)

3.2 Data analysis

This chapter covers our reasoning behind the data collection variables and decisions used in the previous chapter.

3.2.1 Validity and reliability

Implementation

When implementing GraphQL for this study the authors have been looking at implementations from official documentation, the software development community and had discussions with experienced developers. The same has been done when creating the test data and schemas for the two databases in question. The aim was to create a general use case that can be applied to the real-world, but this cannot be promised as many software implementations are highly specialized systems tailored to a specific use case relevant to a specific company.

Research question 1

To ensure validity of our experiments all tests were performed on the same PC. To avoid variables like internet connection the test was performed locally on the machine. This ensures the time measured is purely latency in response time from the system. All code relevant to our experiments is open sourced and can be found on GitHub (Vilhelmsson & Nordström, 2022). For good reliability, the steps taken in each experiment are documented and clearly laid out under the previous experiments subsection, so that anyone attempting to replicate our tests can expect to receive comparable results.

Research question 2

To find out about the reason behind differences or similarities between these databases when using GraphQL we conducted a source code analysis on both database management systems. The source code analysis is available to be conducted by everyone as all systems studied in this paper are open source. The analysis will be conducted extensively in suitable areas with the risk in mind that relevant code could be missed. All conclusions are entirely based on existing code and not speculations.

4 Implementation

In this chapter the different applications used to conduct the Experiments in chapter 3.1.2 will be discussed.

Mentions of file locations in this chapter are in relevance to the root folder of the project available at GitHub.

4.1 Environment

To set up the test environment *Docker* is used. Docker is a platform used to run applications and containers. A docker container is an isolated environment for an application to run, allowing for several applications to be run simultaneously in different containers on the same host (Docker, 2020).

Five different applications are running in four different containers for the conducted experiments to be fulfilled. The Faker application is not running in a container as it is an isolated script that only needs to be run once. Figure 8 below shows applications and the communication between them.

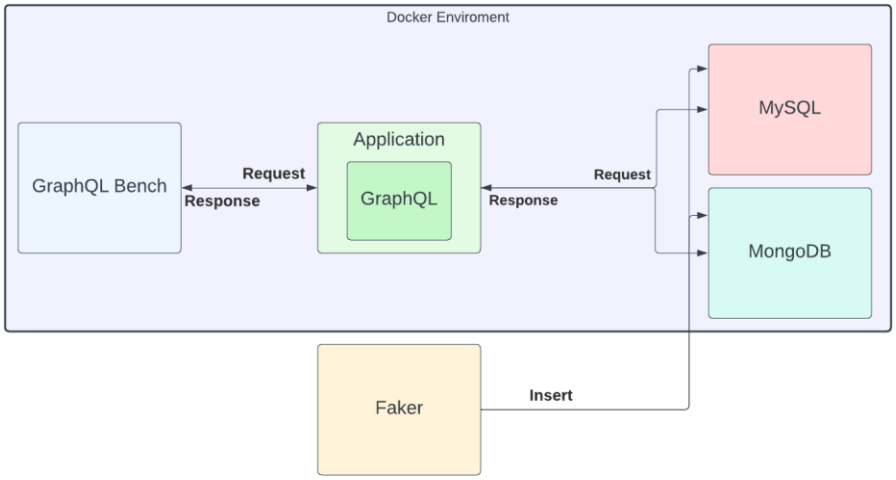

Figure 8: Applications and the communication between them

4.1.1 Application

The application handling the HTTP requests sent by a client is running as an Express web application. Express is a Node.js web application framework (Npm Express, 2022) used to establish the foundation of the web application. In figure 9 on the next page you can see the file structure of the application.

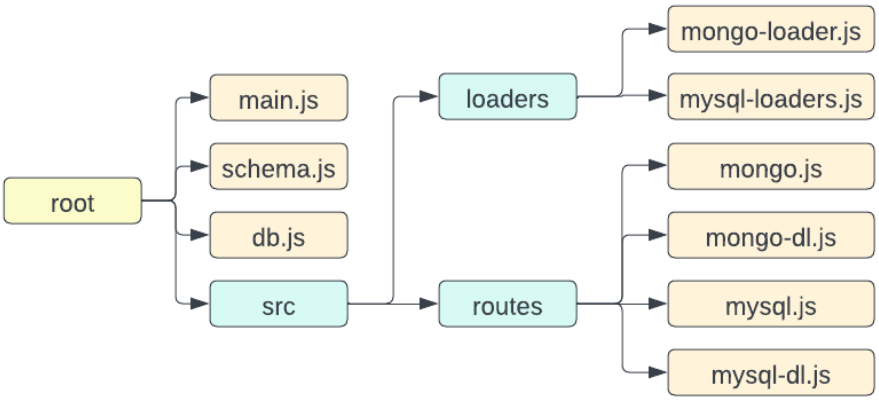

Figure 9: File structure of application

Request Handling

The requests made by GraphQL Bench are handled through routing via the Uniform Resource Identifier (URI), specifying the path in the request. In the example below (figure 10) four different routes are taken depending on what URI is sent.

```
app.use("/mongo", mongoRouter)
app.use("/mysql", mysqlRouter)
app.use("/dl/mysql", mysqlDlRouter)
app.use("/dl/mongo", mongoDlRouter)
```

Figure 10: Request passthrough inside *main.js*

The request is passed from *main.js* into an existing router depending on the URI. The request is then handled within each router individually as seen in figure 11 below.

```
mysqlRouter.use('/', graphqlHTTP({
    schema: buildSchema(librariesSchema),
    rootValue: new Query(),
    graphiql: true
}))
```

Figure 11: Request handler inside *mysql.js*

Routers & Schemas

The routers route to different paths that contain schemas of presented classes. The classes presented are represented by the data objects fetched. *Query* class is the queries that are available to send to the application. *Library*, *Book* and *Librarian* classes represent the different data objects fetched from the databases. Separate routers are made for instances with a DataLoader, used for batching the queries sent by the DataLoader (GraphQL, 2022). In the figure 12 on the next page, the schema is represented.

```
type Library {
  id : ID,
  name : String,
  street : String,
  books: [Book],
  librarians: [Librarian]
}
type Book {
  title : String,
  release_year : Int,
  library_id : ID
}
type Librarian {
  name: String,
  age: Int,
  library_id : ID
}
type Query {
  library(id: Int): Library
  libraries(limit: Int): [Library]
}
```

Figure 12: Schema inside *schema.js*

DataLoader

The routes made with a DataLoader are batching queries when Books and Librarians are fetched with Libraries. The DataLoader waits until several requests have been made to utilize the batching and only hit the database once. In figure 13 below is an example for the Books DataLoader function when using the MySQL DataLoader route.

```
const booksDataLoader = new DataLoader(keys => getBooksByLibraryIds(keys)
{cache: false})
const getBooksByLibraryIds = async (libraryIds) => {
    const query = "select * from books where library_id IN(?)"
    const books = await queryDB(query, [libraryIds]).then(data => data)
    const groupedById = groupBy(book => book.library_id, books)
    const mappedByID = map(libraryId => groupedById[libraryId], libraryIds)
    return mappedByID
}
```

Figure 13: Books DataLoader function inside *mysql-loaders.js*

Database Connection

To establish a database connection from the application to both MongoDB and MySQL read chapter 4.1.2 and 4.1.3.

4.1.2 MySQL

The MySQL docker instance is pulled directly from Docker Hub, and in this project version 5.7.33 was used (mysql:5.7.33). To run queries from the application the npm package “mysql” version 2.18.1 was used. (Npm MySQL, 2019)

Queries run for CRUD (Create, Read, Update, Delete) operations are done according to the example presented in the official npm documentation (figure 14), where a connection first is established, and SQL queries are later passed as arguments to the connection.query() function. In this project the connections are done at start-up in the */application/src/db.js* file and then exported as objects. The queries can then be run by importing the *db.js* file and using the connections from there.

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'my_db'
});

connection.connect();

connection.query('SELECT 1 + 1 AS solution', function (error, results, fie
  if (error) throw error;
  console.log('The solution is: ', results[0].solution);
});

connection.end();
```

Figure 14: Establish MySQL connection (Npm MySQL, 2019)

4.1.3 MongoDB

The MongoDB docker instance is pulled directly from Docker Hub, and in this project the latest version (mongo:5.0.8) was used. To perform CRUD operations on the database we used the npm package “mongodb” version 4.5.0. (Npm MongoDB, 2022)

Operations are done according to example in the official npm documentation (figure 15), to first establish a connection and then perform our operations. In this project the connections are done at start-up in the */application/src/db.js* file and then exported as objects. The operations can then be run by importing the *db.js* file and using the connections from there.

```
const { MongoClient } = require('mongodb');
// or as an es module:
// import { MongoClient } from 'mongodb'

// Connection URL
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

// Database Name
const dbName = 'myProject';

async function main() {
  // Use connect method to connect to the server
  await client.connect();
  console.log('Connected successfully to server');
  const db = client.db(dbName);
  const collection = db.collection('documents');

  // the following code examples can be pasted here...

  return 'done.';
}

main()
  .then(console.log)
  .catch(console.error)
  .finally(() => client.close());
```

Figure 15: Establish MongoDB connection (Npm MongoDB, 2022)

4.1.4 Faker

The npm package to generate data is faker-js/faker. It enables the user to fetch generated data that is made to resemble real-world data.(Npm Faker-js, 2022) Faker is used to generate names, street names, ages and more for values within the objects that are stored in the database. Figure 16 shows an example of how to use Faker and the values chosen for Books and Librarians. Here we push an object into a Books array that has the keys: *title*, *release_year* and *library_id* . The values for *release_year* and *title* are both generated from Faker, *title* has a random hacker phrase and *release_year* has a random number between 1000 and 2022.

```
import {faker} from '@faker-js/faker';

const books = []
const librarians = []

books.push({
    title: faker.hacker.phrase(),
    release_year: faker.datatype.number({min: 1000, max: 2022}),
    library_id: libraryId
})

librarians.push({
    name: faker.name.findName(),
    age: faker.datatype.number({min: 10, max: 100}),
    library_id: libraryId
})
```

Figure 16: Generating data for Librarians and Books

The program in the /faker folder does two things. First, it connects to the two databases (MySQL and MongoDB) and then cleans them from data by dropping all tables and collections and recreating them empty. Second, it generates fake data via Faker and inserts the same data into both databases. This means that when the program is run, the two databases contain the exact same data and every time the program is rerun new data will take its place.

4.1.5 GraphQL-bench

Graphql-bench is a tool for benchmarking and load-testing GraphQL services. The project makes use of version 0.3.1. (GraphQL-Bench, 2022) GraphQL-bench was chosen because of its clear documentation for setup and the ability to easily alter parameters between tests to maintain high reliability for the different cases.

All queries that are run are present in the */graphql-bench/bench.yaml* file, where the test cases described in chapter 3.1.2 Experiments are present. All test cases are run in sequence after the other. GraphQL bench displays the results in graphs after the tests are done, which are presented in chapter 5 Results.

5 Results

In this chapter, the results from the methods used will be presented and analyzed in chapters *Data Collected* and *Data Analysis.*

5.1 Data Collected

5.1.1 Literature Review

Database Complexity

According to research made by Lindvall & Sturesson in 2021 most operations are situated inside the *service_entry_point_common.cpp* file that are connected to reading and fetching data inside the MongoDB database. The analyses they made came with a conclusion that a complexity of O(n) is expected when the read operation is executed by the MongoDB database. O(n) is a linear time complexity for Big-O Notation, where the growth is constant (Dedov, 2020).

The MySQL query complexity is not a simple task to undermine. A comparison made of MySQL and MongoDB for logging text files resulted in O(log(n)) complexity with B-tree indexing for MongoDB and whereas MySQL performance without indexing O(n) (Jandaeng, 2015). MongoDB is of type B-tree, which means it can collect data stored to a certain index or id, called indexing (MongoDB Manual, n.d.). This can also be the case for MySQL if a Primary Key, Foreign Key or specified index key is used while retrieving data (MySQL Documentation, 2019).

The complexity is only a portion of the whole truth, as other methods of query optimization need to be in consideration and not only indexing.

Database Query Optimization

MySQL uses indexes to find rows with specific column values. Instead of starting at the first row and reading throughout the entire table, indexes help to quickly determine the position of data without having to look through all data files. Most of MySQL indexes are stored in B-trees, a self-balancing tree data structure. (MySQL Documentation, 2019) MongoDB also makes use of indexes. Without indexes, MongoDB must perform a collection scan which scans every document in a collection. MongoDB indexes are ordered to support efficient equality matches, and results can be returned sorted using the ordering in the index. (MongoDB Manual, n.d.)

The two different databases MySQL and Mongo handle the query “limit” differently. MySQL fetches all entries before any limit is applied, meaning limit query shall not provide any performance benefits except in retrieval package size. MySQL does provide some optimization if the limit query is paired with for example “group by”. Normally “group by” would be applied first, then the “limit” query is applied after. MySQL optimizes the query “group by” to only group the number of rows specified by the “limit” query (MySQL Documentation, 2019). MongoDB applies the limit query on the cursor that finds all the data before fetching all data. The cursor works by returning pointers to the data locations that shall be fetched, and limit can therefore be applied before any data is fetched. This means the limit query shall in theory have a

positive impact on performance of the query, as it limits the number of documents the cursor must locate and return for fetching. (MongoDB Manual, n.d.)

Jose & Abraham displayed in their research made in 2020 that the query performance for fetching data increases in response times drastically for MySQL when scaling, while MongoDB only increased in response times minimally compared to MySQL. The study used a consistent limit of 25000 for both databases and increased the scaling from 3000 up-to 125000.

5.1.2 Experiments

The following chapter will display the results gathered from the experiments. Each test case will be explained one after the other and comparisons are only done between the different configurations for each test. Meaning, any comparisons in performance is only done in comparisons between the performance results gathered from that specific test case. The graphs display performance in response time, therefore lower values equal better performance and display the average value for the duration of the test period.

Parameters

The tests are done using GraphQL bench. The tests can be configured using parameters that we will mention under each test. Some parameters that can be changed by the user are explained below, and some of them change during the tests.

Requests per second (RPS)

How many requests the benchmarking tool is requesting per second.

Warmup duration

A set time where the benchmarking tool is requesting data from the application but not accounting for them in the results.

Duration

The time the benchmarking tool is requesting data from the application. The response time of these requests are the results presented and discussed.

Open connections

Number of connections sending the requests to the database. The number used for all tests is 20 as it was the standard value.

Test Cases 1-6

The test is done with warmup duration of 20 seconds, a duration of 60 seconds, with first 100 requests per second (RPS) and then 200 RPS. This means that the test will run twice for each test case, and the results presented are both with 100 RPS and 200 RPS.

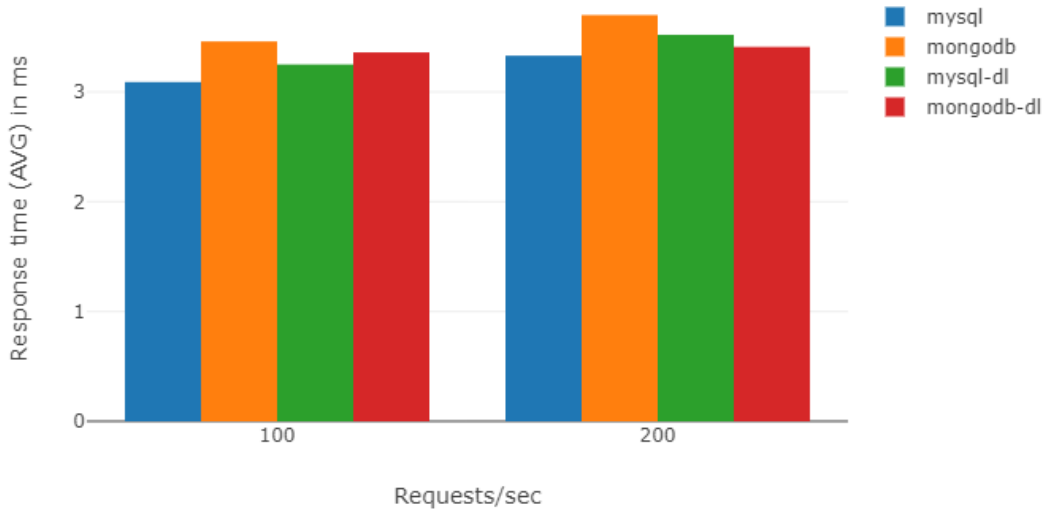

Figure 17: Test Case 1

The data from Test Case 1 (figure 17) fetches without any relations between datasets, therefore the DataLoader is not used. This should lead to comparable results between the two databases, and as seen in the results, this is the case. MongoDB is slightly slower than MySQL, both in 100 RPS and 200 RPS. The DataLoaders for both MongoDB and MySQL are also comparable, where the MongoDB DataLoader slightly outperforms MySQL DataLoader at 100 RPS, but the results are in favor of MySQL DataLoader at 200 RPS. Overall, MongoDB is slower than MySQL and the DataLoader for MongoDB improves response time while DataLoader for MySQL worsens response time.

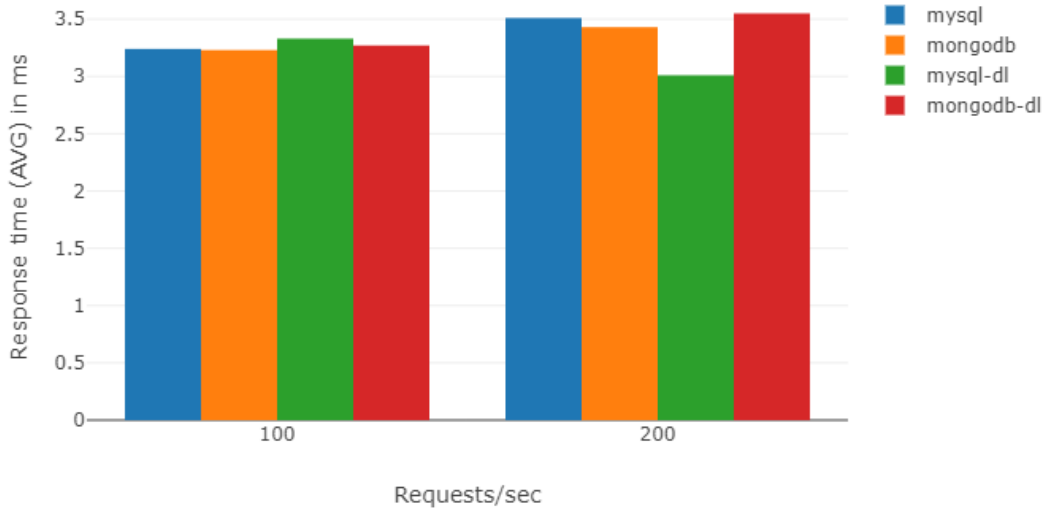

Figure 18: Test Case 2

The data in Test Case 2 (figure 18) fetches 1000 Books per Library fetched. This means that DataLoaders are batching the queries for Books until all book ids belonging to a Library is found out before sending a query to fetch them. The results at 100 RPS are

comparable across all four configurations. The results at 200 RPS show some difference where MySQL has a slower response time than MongoDB. MySQL with DataLoader is the fastest and MongoDB with DataLoader is the slowest regarding response times.

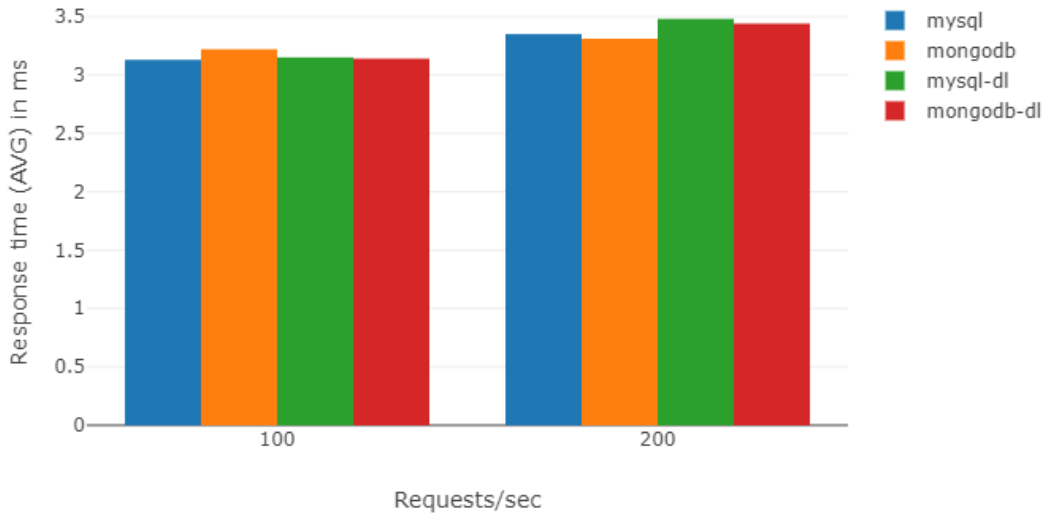

Figure 19: Test Case 3

The data fetched in Test Case 3 (figure 19) is the same as in Test Case 2, but with 1000 Librarians per Library added. The results in Test Case 3 are all comparable to each other. There are marginal differences when RPS is set to 200, showing the with DataLoaders are slightly slower than without DataLoader.

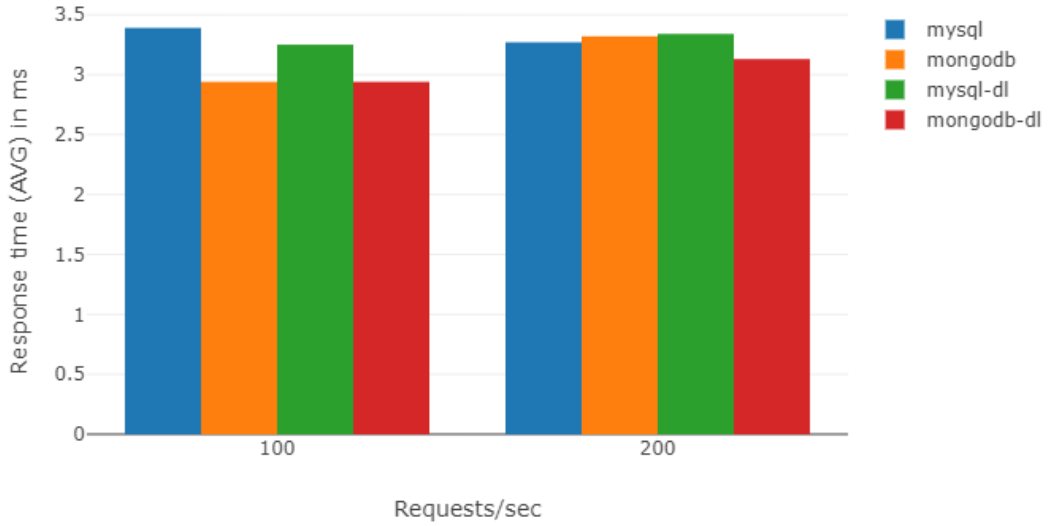

Figure 20: Test Case 4

The data fetched in Test Case (figure 20) 4 is the same as in Test Case 1, but with 10 libraries instead of just 5 libraries. At 100 RPS, MySQL is the slowest and MySQL

with DataLoader comes after that. The results from MongoDB with and without DataLoader are very similar, both faster than MySQL. The response times for 200 RPS show marginal differences between MongoDB and MySQL (with and without DataLoader). The fastest one is MongoDB DataLoader.

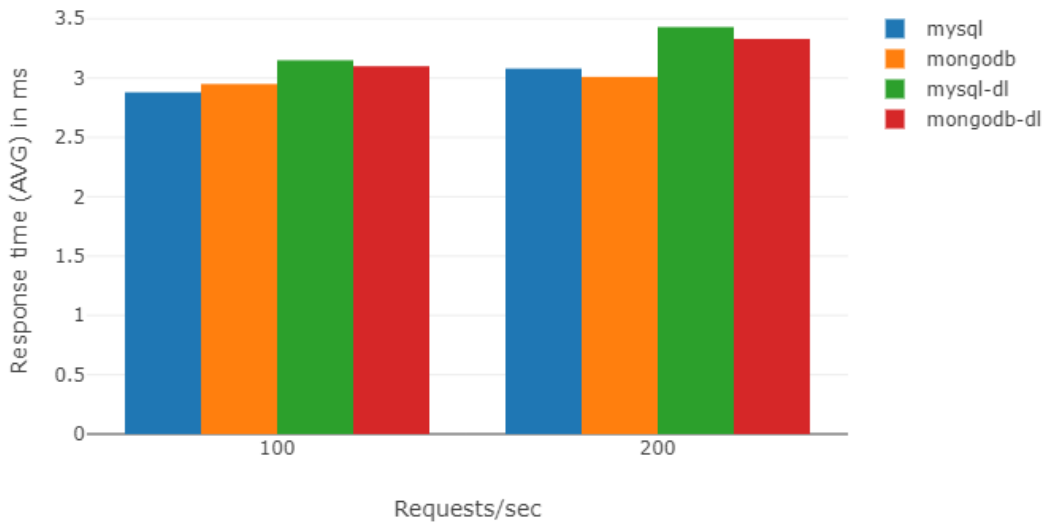

Figure 21: Test Case 5

The data fetched in Test Case 5 (figure 21) is the same as in Test Case 2, but with 10 libraries instead of just 5 libraries. The results displayed between MySQL and MongoDB show similar results when DataLoader is not in use. DataLoaders response times are worse both for 100 and 200 RPS compared to without, where the MySQL DataLoader performed the worst.

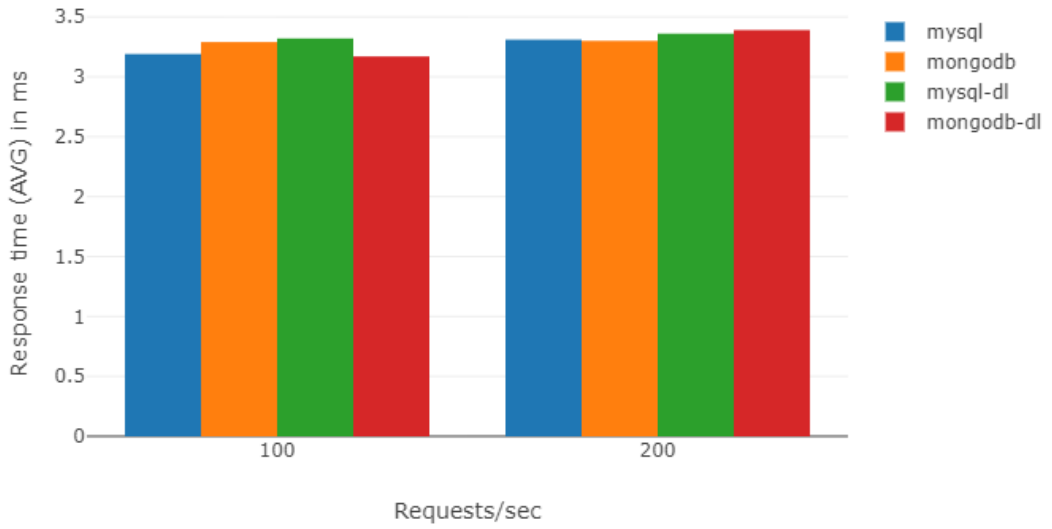

Figure 22: Test Case 6

The data fetched in Test Case 6 (figure 22) is the same as in Test Case 3, but with 10 libraries instead of just 5 libraries. The results in Test Case 6 show similar response times with only marginal differences between the configurations. At 100 RPS MongoDB and MySQL DataLoader are slightly slower. At 200 RPS MySQL DataLoader and MongoDB DataLoader are slightly slower than the databases without DataLoaders.

Test Cases 7-12

The test uses the same configuration parameters as previous tests. What differs Test Cases 1-6 from Test Cases 7-12 is the amount of data fetched. In Cases 1-6, 1000 rows of Books and Librarians were fetched per Library, and in Test Cases 7-12, 20 000 rows of Books and Librarians were fetched per Library.

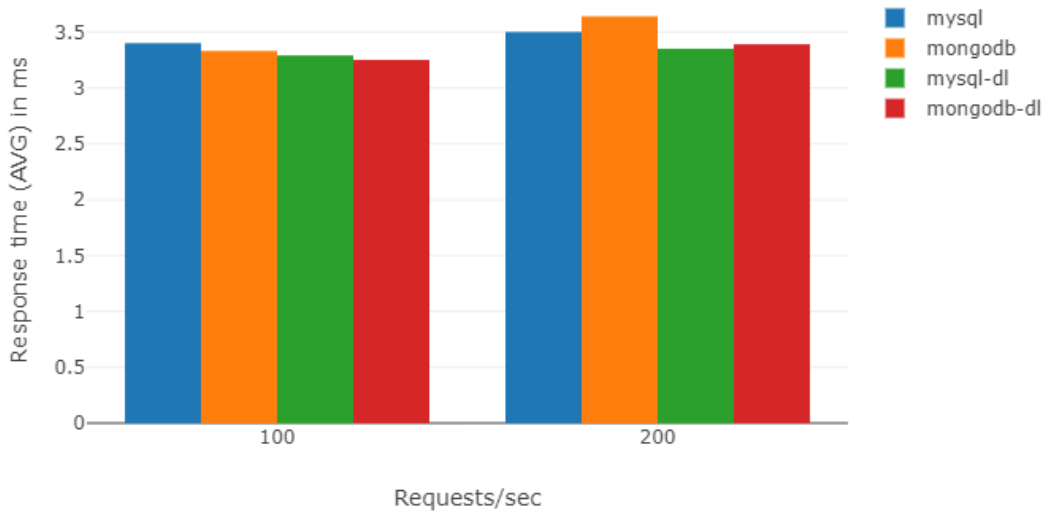

Figure 23: Test Case 7

The data Test Case 7 (figure 23) shows a small difference between the four configurations at 100 RPS, where DataLoaders do perform better. At 200 RPS, MongoDB is slightly worse compared to the others. Performance is better with DataLoaders for both MySQL and MongoDB at 100 and 200 RPS.

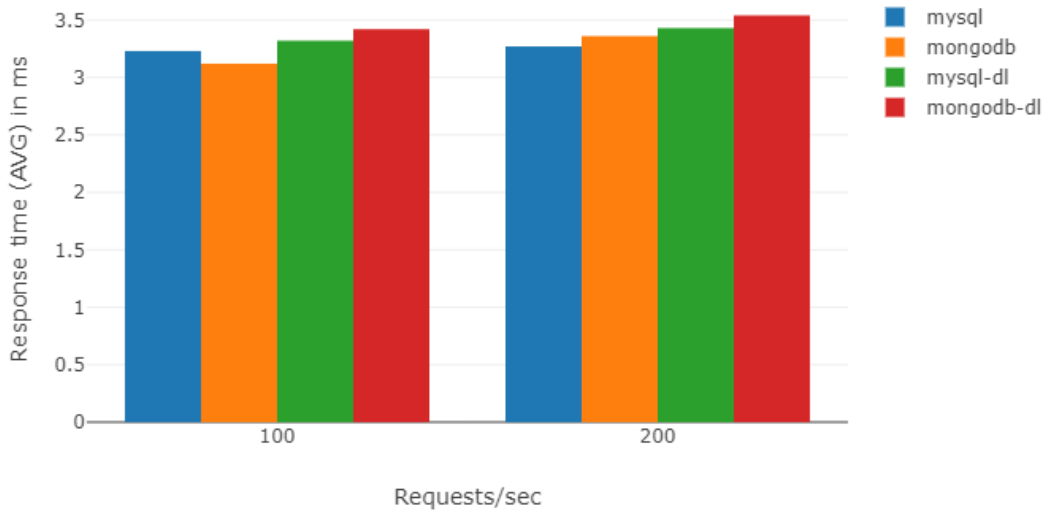

Figure 24: Test Case 8

The data from Test Case 8 (figure 24) displays similar results between 100 and 200 RPS between DataLoaders, while the performance is a bit worse using higher RPS for MongoDB. DataLoaders perform worse in both RPS tests.

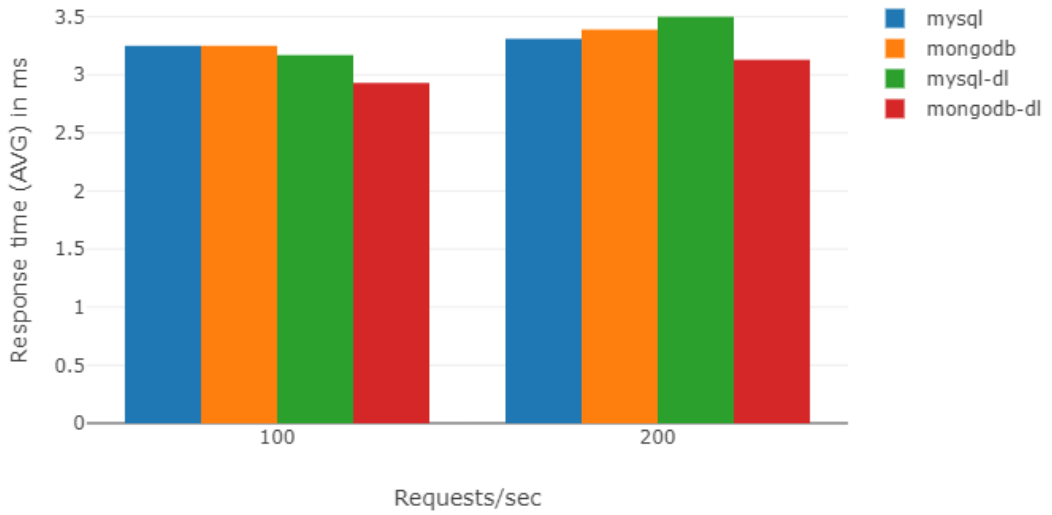

Figure 25: Test Case 9

The data from Test Case 9 (figure 25) at 100 RPS displays no difference between MySQL and MongoDB without a DataLoader. MongoDB with DataLoader was fastest with a good margin. At 200 RPS MongoDB DataLoader is the fastest and maintains the response time, while MySQL DataLoader is the slowest. MySQL and MongoDB are comparable to each other.

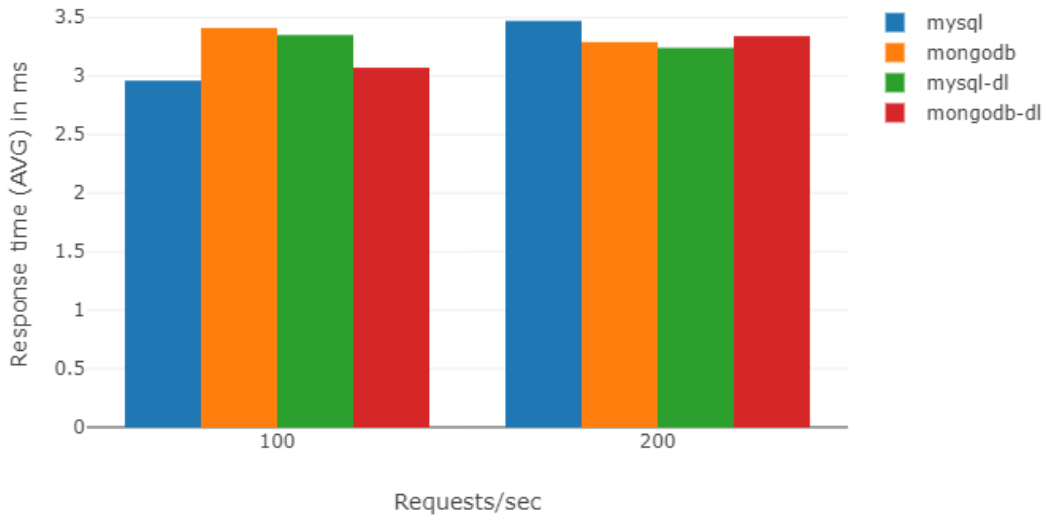

Figure 26: Test Case 10

In Test Case 10 (figure 26) MySQL's response times is the fastest at 100 RPS but the slowest at 200 RPS. Making use of the DataLoader MySQL then becomes the fastest at 200 RPS compared to other configurations. The response times for MongoDB is worse when not using a DataLoader at 100 RPS but becomes similar when using DataLoader at 200 RPS.

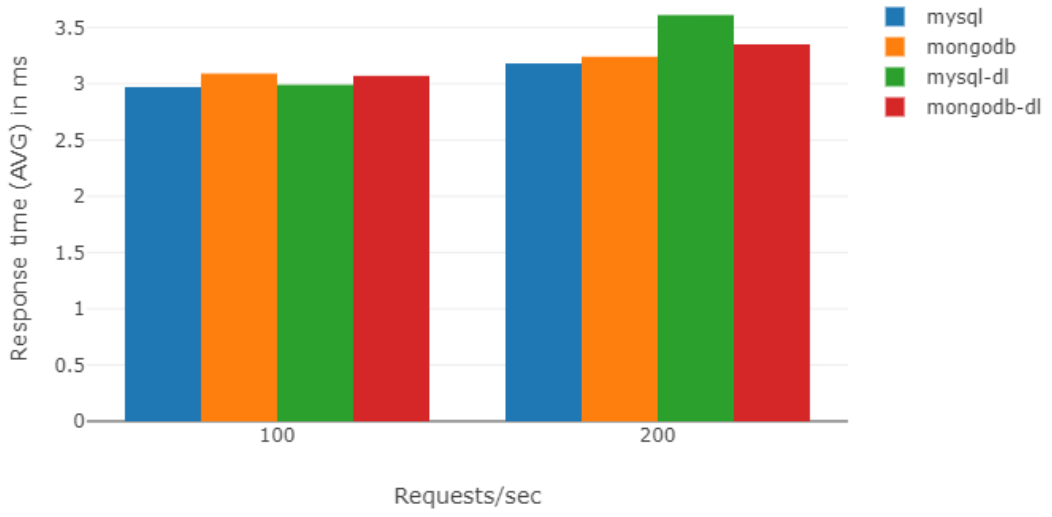

Figure 27: Test Case 11

The results for Test Case 11 (figure 27) show little distinction between the four different configurations at 100 RPS. At 200 RPS MySQL is marginally faster than MongoDB, but MySQL with DataLoader is slower compared to MongoDB with DataLoader. At 100 RPS the performance between DataLoaders and without is equivalent across the different configurations.

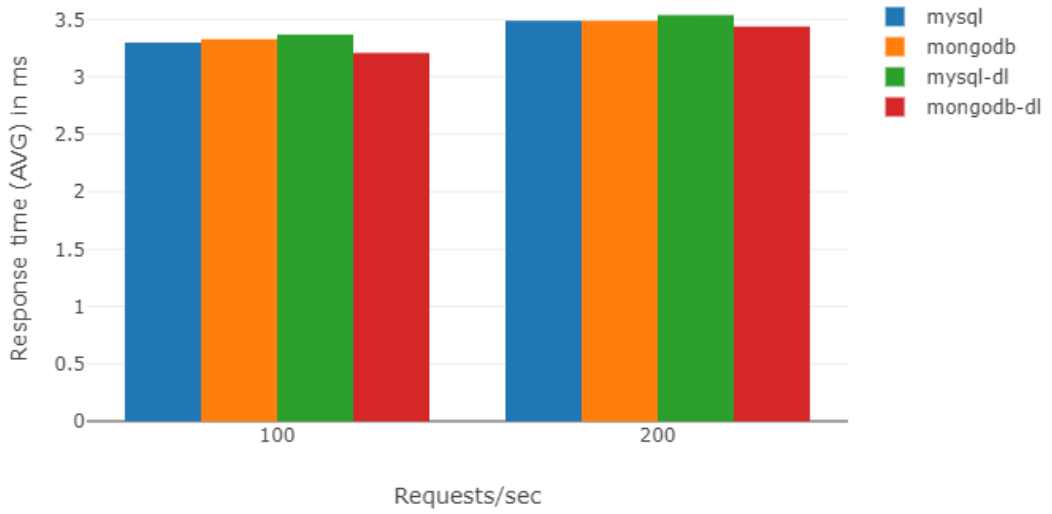

Figure 28: Test Case 12

The differences in response times in Test Case 12 (figure 28) are minimal and the performance is overall worse when RPS is 200. The fastest across both tests was MongoDB with DataLoader by a small margin.

Test Cases 13-18

The test cases use a different configuration, with 500 RPS instead of 100 and 200 RPS. The amount of data fetched is the same as in Test Cases 1-6, 1000 rows per Library.

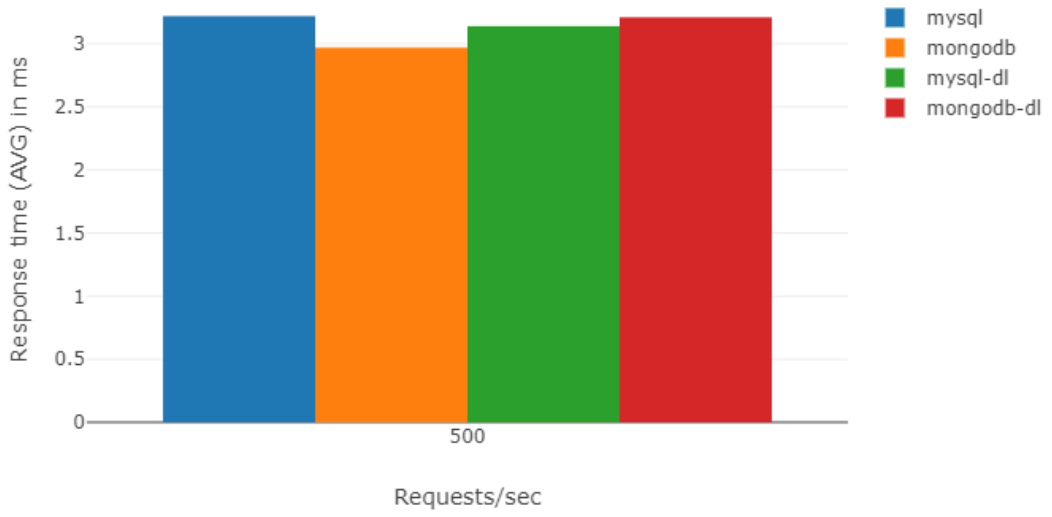

Figure 29: Test Case 13

The results from Test Case 13 (figure 29) show a performance improvement for MySQL when using DataLoader while MongoDB performance is worse when paired

with DataLoader. MongoDB is faster than MySQL when the databases are not paired with DataLoaders.

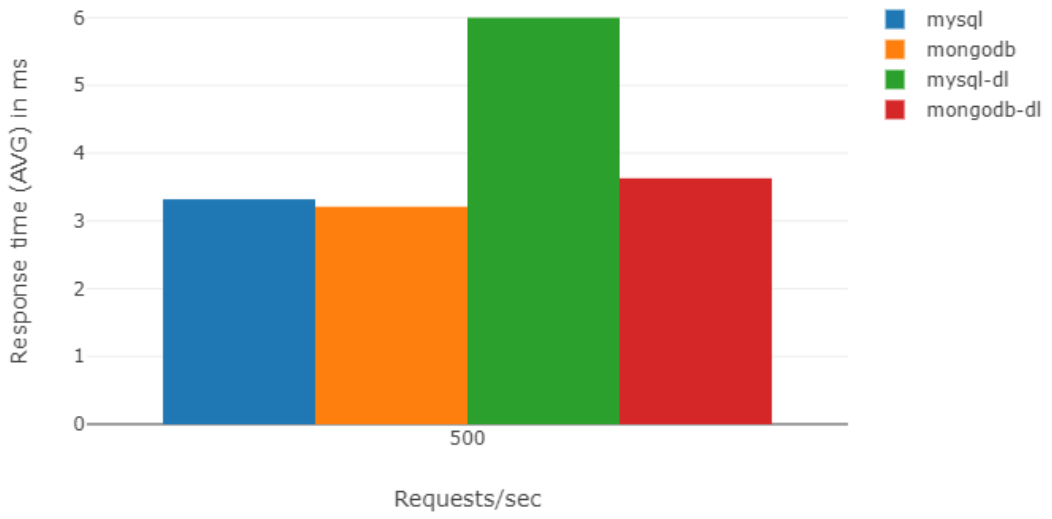

Figure 30: Test Case 14

In Test Case 14 (figure 30) there is a clear performance difference between MySQL and MySQL with a DataLoader due to the response time almost doubles. MongoDB with DataLoader is also slightly worse than MongoDB.

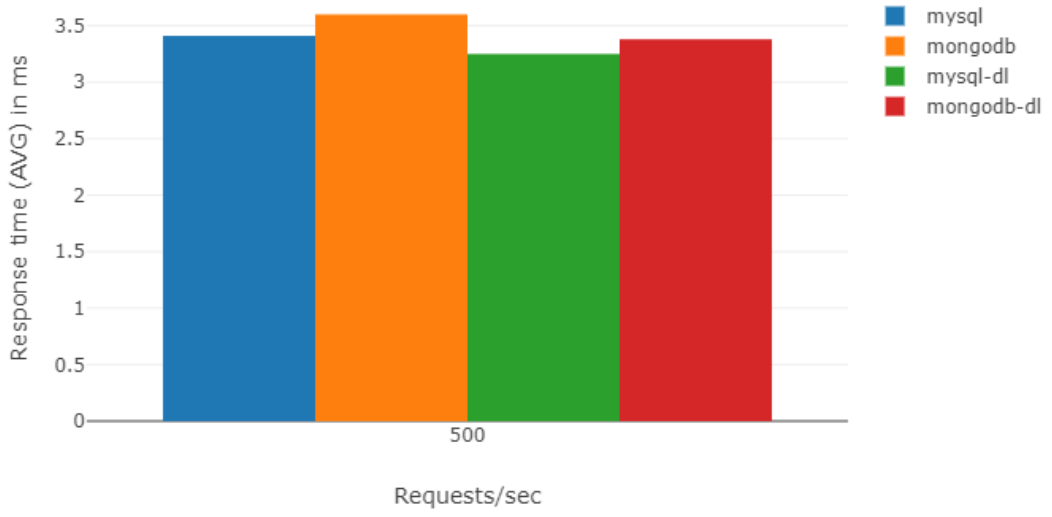

Figure 31: Test Case 15

Results from Test Case 15 (figure 31) display a marginal performance improvement using DataLoaders compared to without for both databases. MySQL has slightly faster response time compared to MongoDB both with and without a DataLoader.

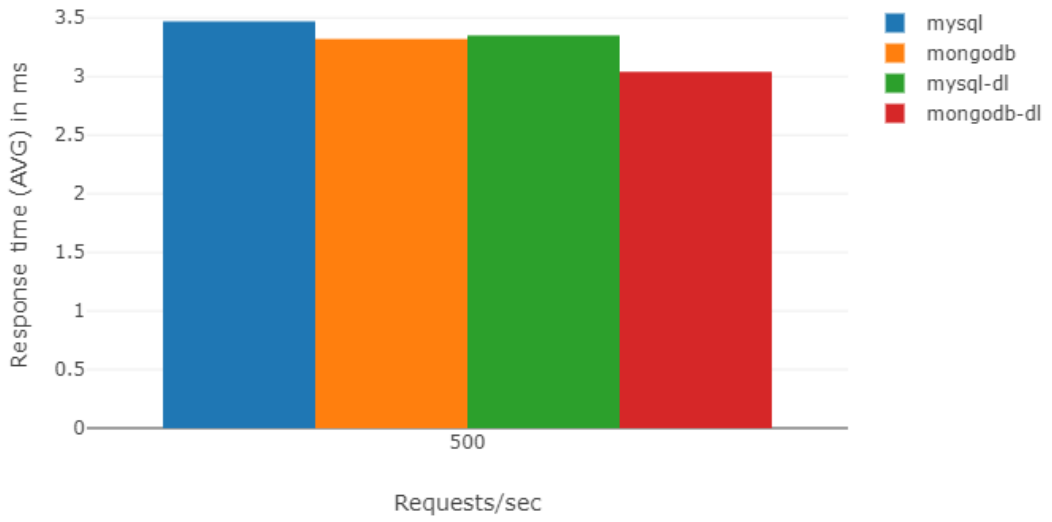

Figure 32: Test Case 16

The results in Test Case 16 (figure 32) display that MySQL is slightly slower than MongoDB. With DataLoaders the results are the same except lower in response time, MySQL being slightly slower than MongoDB.

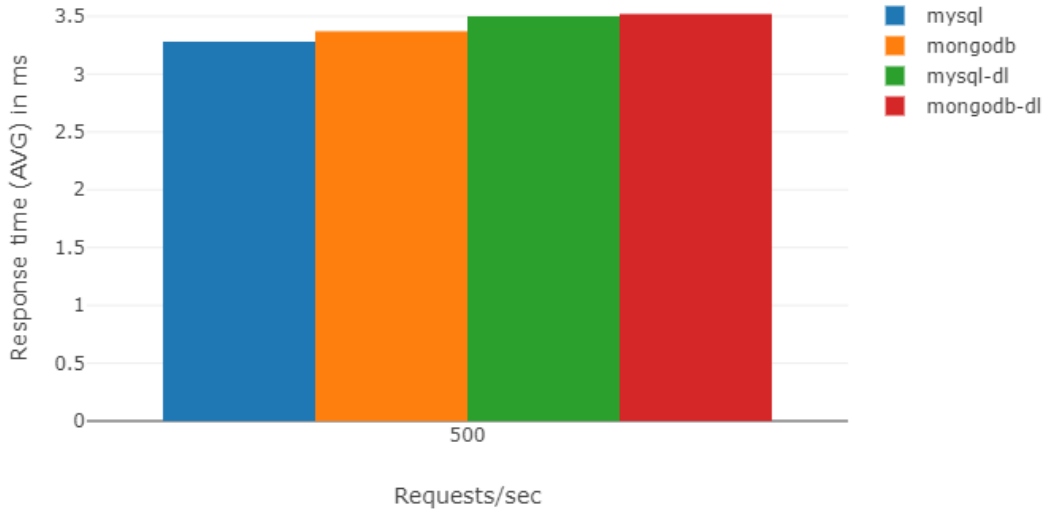

Figure 33: Test Case 17

The data from Test Case 17 (figure 33) show no major differences between configurations. Databases with DataLoaders performed marginally worse compared to without, with MySQL having the fastest response time.

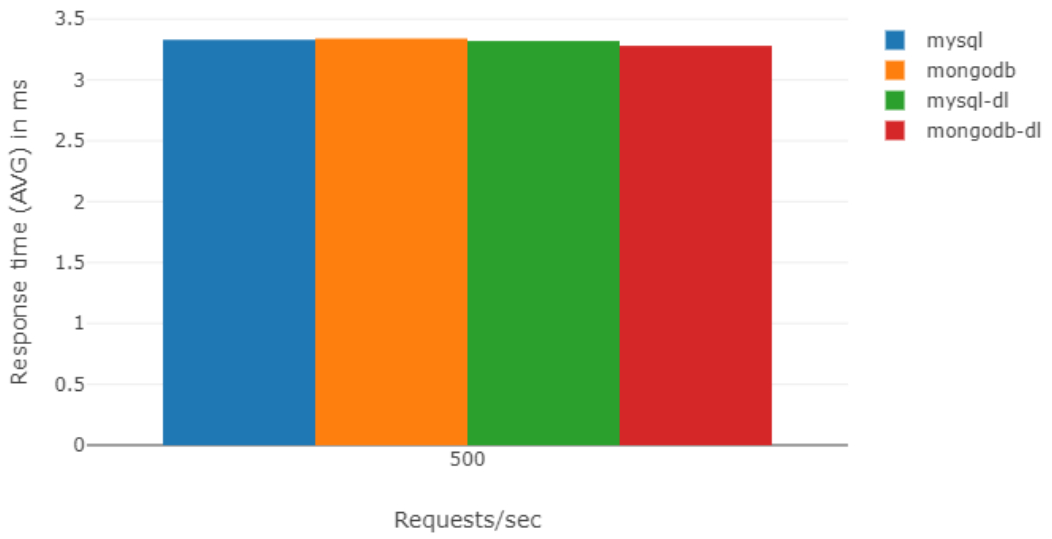

Figure 34: Test Case 18

The differences in response time in Test Case 18 (figure 34) are minimal to none, however DataLoaders do perform better.

5.2 Data Analysis

5.2.1 Literature Review

MySQL and MongoDB both take advantage of the B-tree structure and indexing, which means that the retrieval times will differ from O(log(n)) to O(n). As both databases share a similarity of complexity, an assumption could be made that the differences come from the optimization of the query and indexing instead.

The indexing differences between MongoDB and MySQL boils down to how the structure of the data is being made. MySQL's tables Librarians and Books contains the ID of a Library as a Foreign Key constraint relation, essentially generating that as an index for quicker look-up. MongoDB does not do this but rather solely depend on the key as a value.

Scaling the databases also affects the query performance, as increased the scale proves to be a disadvantage for MySQL compared to MongoDB. A cause for this could be how the limit operation functions differently on the databases and is not as efficient on MySQL compared to MongoDB.

5.2.2 Experiments

Scenario 1 with cases 1, 7 and 13

The difference between cases 1, 7 and 13 is interesting because there is no DataLoader active working to batch queries between entries. However, the results show that in general the DataLoader either improves the response time or it stays the same. The results also display an interesting correlation between fetching times when the databases contain 1000 entries for Books and Librarians per Library compared to 20000 entries. While MySQL's performance worsens with more entries to the database,

MongoDB remains almost the same. MongoDB's performance even improves when the requests sent per second reaches 500 compared to the 100 and 200.

Scenario 2 with cases 2, 8 and 14

The difference between cases 2, 8 and 14 does not show a favor towards either configuration. In test case 2 MongoDB shows no real difference with or without a DataLoader, and MySQL with DataLoader is only faster when the test is run with 200 RPS. In test case 8 a general trend of DataLoaders being slower than without is shown. In test case 14 we have an outlier with MySQL with DataLoader clearly being much slower than the rest, and MongoDB with DataLoader is slightly slower than without DataLoader. The outcome of scenario 2 is that DataLoader is marginally slower compared to without a DataLoader when one join is used.

Scenario 3 with cases 3, 9 and 15

The difference between cases 3, 9 and 15 does not provide a clear trend towards either configuration. MongoDB paired with a DataLoader performs better than without in test case 9, and it is the opposite for MySQL. Test case 3 shows marginally slower results for DataLoader only at 200 RPS, and in test case 15 DataLoader shows better results. The outcome of scenario 3 is that DataLoader is marginally faster or at least equal compared to without a DataLoader.

Scenario 4 with cases 4, 10 and 16

Comparing the results from cases 4, 10 and 16 a trend can be seen where MongoDB with and without DataLoader is faster than MySQL with and without a DataLoader. The outlier is test case 10, where MySQL without a DataLoader is the fastest at 100 RPS. The impact the DataLoader has in this scenario is an improvement across the board. While it is marginally faster compared to without a DataLoader, it still is faster in almost all cases. These results are interesting due to the fact that during these test cases the DataLoaders shall not provide a difference as they never batch any queries.

Scenario 5 cases 5, 11 and 17

The results display that GraphQL with DataLoaders perform equal or worse in all cases. At test case 11 the results for 100 requests-per-second (RPS) show little to no difference at all, and at 200 RPS the DataLoaders performance worsens especially for MySQL.

Having comparable results between 100 and 200 RPS from test case 5, and 500 RPS from test case 17 show a clear disadvantage to using a DataLoader.

Scenario 6 with cases 6, 12 and 18

Comparing the results from test cases 6, 12 and 18 it is evident that MongoDB using a DataLoader do perform generally better than without one in most cases. MySQL does quite the opposite, as its query performance worsens when using the DataLoader. With that said, the DataLoader do both perform minimally better when the requests-per-second reaches 500 compared to 100 and 200.

6 Discussion

In this discussion chapter the *Result* and *Method* will be discussed.

6.1 Result discussion

The results from scenario 1 and 4 in chapter 5.2.2 displays interesting results. The test cases in scenarios 1 and 4 are only fetching libraries and does not join Librarians and Books. This means that the DataLoaders are not active when they batch queries for related data to each entry fetched. Still, MySQL and MongoDB paired with DataLoaders are generally faster than without DataLoaders. This most likely depend on how the application compiles the code and handles the request rather than the actual query.

It is the opposite for scenario 2, where the DataLoaders are activated to batch queries for fetching Books belonging to each Library. Still, the outcome is that DataLoaders are generally slower compared to without.

At test cases 9, 11 and 14 MySQL with DataLoader is substantially slower compared to the other configurations. What these test cases have in common is that at least one join is performed by GraphQL to fetch the needed data, and all test cases make use of a DataLoader. The query optimization for MySQL after selecting rows for multiple indexes while looking for a given set of keys seems worse while using a DataLoader, indicating that the query optimization here is better when using MongoDB even though MySQL has a B-tree of indexes containing foreign keys.

Scenario 5 shows that only using one join through GraphQL is a disadvantage when a DataLoader is in use. While scenario 6 displays joining two datasets is a general improvement for MongoDB but not MySQL. This is an indication that MySQL optimizes and selects data better than GraphQL with a DataLoader in combination with MySQL does in most cases.

Comparing solely MongoDB against MySQL when using GraphQL without a DataLoader the results display no indication of a pattern between test cases. Marginal improvements and deterioration can only be seen as how the query was optimized by the database and not affected by the GraphQL solution.

6.2 Method discussion

The structure for schemas used in this thesis work was inspired by Erlandsson & Remes (2020). Their benchmark comparisons show drastic differences for response time in disfavor towards the GraphQL API. The authors of this report wanted to recreate a similar experiments with focus on GraphQL to evaluate its performance with different underlying architectures. Therefore, the test cases where design to mimic the amount of column joins used by Erlandssons and Remes with similar schema structure. The database structure for MySQL and MongoDB is different but the schemas are made to be comparable. Schema design could be made more effective with documents and sub-documents (embedded documents) for relational data when using MongoDB.

Parameters for GraphQL Bench provide the ability to change requests per second (RPS) and the duration time of the benchmark. The request per second (RPS) at 100, 200 and

500 were chosen to see if any difference in response time was present when using different RPS. With a duration of 60 seconds, it results in 6 000, 12 000 and 30 000 requests made to databases under a test case to supply a good sample size for an average response time. Further research could be done with different RPS and durations for a different sample size.

DataLoader has two main features, batching queries (tested in this thesis) and caching data. The caching of data was disabled for all test cases. With caching enabled it would mean only the first run in each test case would give data relevant to the thesis, the main goal of analyzing the retrieval times from databases. Caching simply would not be compatible with the chosen test cases as cached data would be reread instead of fetched again. This is an important limitation of the experiments, as it limits the potential performance gain of DataLoader.

7 Conclusions and further research

7.1 Conclusions

The results from this study show that the difference between using a MongoDB database solution compared to MySQL with GraphQL had minimal difference in query response times. In general, the results do not favor one solution over the other, and the marginal difference between test-cases could potentially be seen as a margin of error for the performance seen. The response times show vague and inconsistent patterns between test-cases. Therefore, conclusions regarding what differ between the retrieval times is more likely the query optimization of the databases and the implementation of the GraphQL solution, not the database complexity. Since both databases have a Big-O Notation from O(log(n)) to O(n) depending on how the request is handled by the database itself.

What impacts the query performance is the query optimization and implementation of the GraphQL solution. As the joins are handled through GraphQL, the integrated joins existing for MySQL are not being used. This is evident when using MySQL with a GraphQL solution and a DataLoader as retrieval times can sometimes be unreliable and slower compared to without a DataLoader. This is the cause of batching the request instead of sending multiple requests to the database, which for MySQL is faster in most cases in the experiments made. This is instead more favorable for MongoDB as the query batching speeds up the look-up in the database and increases performance the more requests-per-seconds is being used.

7.2 Further research

The limitations of this study leave room for further research. As mentioned in Method Discussion, chapter 6.2, parameters for the benchmarking tool can be altered for a different sample size. A larger dataset set could result in a bigger difference between the databases chosen in this study, and the scalability of databases should be considered when doing so.

Further evaluation of the GraphQL DataLoader could be done whilst using caching as this could lead to different results. Test cases could be designed for comparison with caching to get greater insight into the impact a DataLoader has (regarding response time) for different database solutions.

Furthermore, the experiments are performed with local instances but could be done with remote servers to add the factor of internet traffic to the results of response times. This would increase the real-world application of the experiments. The databases in this study could also be exchanged for other types of database management systems.

The database complexity discussed in chapter 5.1.1 and 5.2.1 along with the source code could be further investigated for both MongoDB and MySQL, as it would lead to a stronger argument about the equality or inequality of the structural differences between the databases.

8 References

Assaf, W. (2022, June 23). *Databases - SQL Server*. Docs.microsoft.com. Retrieved August 28, 2022, from https://docs.microsoft.com/en-us/sql/relational-databases/databases/databases?view=sql-server-ver15

Biehl, M. (2015). *API Architecture: The Big Picture for Building APIs* (Vol. 2). API-University Series.

Cronin, M. (2019, August 1). *What is the N+1 Problem in GraphQL?* The Marcy Lab School. Retrieved August 28, 2022, from https://medium.com/the-marcy-lab-school/what-is-the-n-1-problem-in-graphql-dd4921cb3c1

Dedov, F. (2020). *The Bible of Algorithms and Data Structures: A Complex Subject Simply Explained (Runtime Complexity, Big O Notation, Programming)*. Independently published.

Docker. (2020, April 9). *Docker overview*. Docker Documentation. Retrieved August 28, 2022, from https://docs.docker.com/get-started/overview/

Erlandsson, P., & Remes, J. (2020). Performance comparison : Between GraphQL, REST & SOAP. In *his.diva-portal.org*. University of Skövde. http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-18713

Express. (2022). Npm Express. Retrieved August 28, 2022 from https://www.npmjs.com/package/express

Faker-js/faker. (2022). Npm Faker-Js. Retrieved August 28, 2022, from https://www.npmjs.com/package/@faker-js/faker

Geene, M., Busch, B., & Jenkins, L. (2022). *The State of API integration*. Cloud Elements. https://f.hubspotusercontent40.net/hubfs/440197/Cloud-Elements-2021-SOAI.pdf

GraphQL. (2022). *DataLoader*. GitHub. Retrieved August 28, 2022 from https://github.com/graphql/dataloader#readme

Hazura. (2022, May 25). *GraphQL Bench*. GitHub. https://github.com/hasura/graphql-bench

IBM. (2010). *What is a database management system?* Www.ibm.com. https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-what-is-database-management-system

IBM Cloud Education. (2020, August 19). *What is an Application Programming Interface (API)?* Retrieved August 28, 2022 from https://www.ibm.com/cloud/learn/api

Ince, D. (2019). *A Dictionary of the Internet* (D. Ince, Ed.). Oxford University Press. https://doi.org/10.1093/acref/9780191884276.001.0001

Jandaeng, C. (2015). Comparison of RDBMS and document oriented database in audit log analysis. *2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE)*. https://doi.org/10.1109/iciteed.2015.7408967

Jose, B., & Abraham, S. (2020). Performance analysis of NoSQL and relational databases with MongoDB and MySQL. *Materials Today: Proceedings*, *24*, 2036–2043. https://doi.org/10.1016/j.matpr.2020.03.634

JSON. (n.d.). Retrieved August 28, 2022 from https://www.json.org/json-en.html

Lindvall, J., & Sturesson, A. (2021). *A comparison of latency for mongodb and PostgreSQL with a focus on analysis of source code*. DIVA.

Lucidchart. (n.d.). *What is a Database Schema*. Lucidchart. Retrieved August 28, 2022, from https://www.lucidchart.com/pages/database-diagram/database-schema

MongoDB. (n.d.). *Introduction to MongoDB*. MongoDB Manual. Retrieved August 28, 2022 from https://docs.mongodb.com/manual/introduction/

MongoDB. (n.d.). *Model One-to-Many Relationships with Embedded Documents*. MongoDB Model Relationships. Retrieved August 28, 2022 from https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/

MongoDB. (n.d.). *What is a Document Database?* MongoDB Document Databases. Retrieved August 28, 2022 from https://www.mongodb.com/document-databases

MongoDB. (2022, August 18). Npm MongoDB. Retrieved August 28, 2022 from https://www.npmjs.com/package/mongodb

MySQL. (2019). Npm MySQL. Retrieved August 28, 2022 from https://www.npmjs.com/package/mysql

MySQL. (2019). MySQL Documentation. Mysql.com. Retrieved August 28, 2022 from https://dev.mysql.com/doc/

Patni, S. (2017). Fundamentals of RESTful APIs. *Pro RESTful APIs*. https://doi.org/10.1007/978-1-4842-2665-0_1

Prisma. (n.d.). *GraphQL with Database & Prisma | Next-Generation ORM for SQL Databases*. Prisma. Retrieved August 28, 2022 from https://www.prisma.io/graphql

Rinquin, A. (2017, July 6). *Avoiding n+1 requests in GraphQL, including within subscriptions*. Slite. Retrieved August 28, 2022 from https://medium.com/slite/avoiding-n-1-requests-in-graphql-including-within-subscriptions-f9d7867a257d

Stephenson, N. (1999). *In the beginning ... was the command line*. Avon Books.

The GraphQL Foundation. (n.d). *GraphQL: A query language for APIs*. Graphql.org. Retrieved August 28, 2022, from https://graphql.org/

Vilhelmsson, M., & Nordström, D. (2022). *Didrik-Marcus-Thesiswork*. GitHub. Retrieved August 28, 2022, from https://github.com/Didrik-Marcus-Thesiswork

Vogel, M., Weber, S., & Zirpins, C. (2018). Experiences on Migrating RESTful Web Services to GraphQL. *Service-Oriented Computing – ICSOC 2017 Workshops*, *10797*, 283–295. https://doi.org/10.1007/978-3-319-91764-1_23

Wieruch, R. (2018). *The road to GraphQL : With React Class Components*. Robin Wieruch.

Wong, C. (2000). *HTTP pocket reference*. O'Reilly.