Optimization and Parallelization of Object-Relational Mappers

Filip Jovanov*, Vladimir Zdraveski*, Marjan Gusev*, Magdalena Kostoska*
* Faculty of Computer Sciences and Engineering, Skopje, North Macedonia
filip.jovanov@students.finki.ukim.mk, {vladimir.zdraveski, marjan.gushev, magdalena.kostoska}@finki.ukim.mk

***Abstract*—Most of today's software is connected to the Internet, no matter its type/usage. The developers of these applications constantly have to make important decisions that impact the software's performance. One predominant area of these discussions is whether to use Object Relational Mappings or native SQL. In this paper, we introduce a new Java library called RQL and an algorithm to drastically optimize ORMs, specifically time and memory complexity, as well as the number of database calls that they produce. We set a hypothesis to check if this new approach performs better than the conventional one. The evaluation of conducted experiments proves that the new approach based on the division of work to separate threads, proper decision-making, partitioning, in-memory data mapping, entity pre-processing and the avoidance of Cartesian products and n+1 issues achieves better performance and reduces resource requirements.**

Keywords*—*object-relational mappers, entity preprocessing, partitioning, distributed processing

I. Introduction

Our management specified a task to develop a solution based on a lot of legacy code, while also implementing new features. Around the end of development, management wanted a new functionality that was akin to GraphQL. The development team compared the remaining time to develop the new functionality with the estimated time of refactoring the whole codebase to a GraphQL implementation. We determined that it wasn't feasible.

So, we implemented a similar functionality that would meet the bare requirements and would be within the available time frame. After implementing it, we noticed that it brought a significant performance issue, especially regarding database access.

We then looked at different options to improve the performance and decided on creating a number of utility methods which we afterward turned into a library.

In this paper, we introduce said library, as well as explain the logic and decision-making algorithms related to it, along with its possibilities for parallel processing.

One real-world use case for this library is access to aggregation entities and data transfer objects (DTOs) like in a software product used by a government agency that regulates and reports on the systematization of other government agencies, or the annual salaries. These reports usually consist of many joins and a huge amount of data.

In a relational database, this can pose an issue, depending on the implemented solution like creating N+1 issues, cartesian products, technical debt, or restricting possible future functionality.

We aim to prove the following research hypothesis: *Time complexity, memory complexity, and the number of requests being sent to a database can be improved by disabling the possibility of Cartesian products and n+1 issues at run-time, as well as dividing the work into smaller jobs dealt by separate threads using a variety of different techniques.*

To prove the validity of this hypothesis we measure the response times and the counts of multiple different types of database requests, with multiple different data sets. The measurement is realized using semi-standard REST and standard GraphQL implementations, with simple JPA repositories, then, modifying the semi-standard REST implementation to use the synchronous RQL library, and finally the parallelized RQL library. Measured data are collected using Python scripts to measure the response times and Java methods to collect the number of database calls detected by P6Spy.

The rest of the paper is organized in the following structure. Section II presents similar solutions like GraphQL and Join-Monster, as well as key packages that RQL uses and an overview of similar useful papers. Section III elaborates on the Cartesian product and n+1 issues, undocumented Spring Data JPA relational rules, as well as Object Relational Mappings (ORMs) stances toward optimization. Then we dive into our library's solution, explaining how the entity pre-processing, partitioning, mapping, and parallel processing are implemented in Section IV. Analysis of all the different test cases and payloads, as well as the results and their evaluation, is presented in Section V. Finally, Section VI concludes the paper and gives directions for future works.

II. Related Work

One of the huge problems in the development of backend services is the huge amount of effectively equivalent REST endpoints, either due to constantly changing requirements and not wanting to break previous implementations or because the problem is structured in such a way that the same entity needs to be displayed in multiple different ways. The current solution to this problem is GraphQL [1]. GraphQL allows one to "pick and choose" what is needed, by accessing a single endpoint.

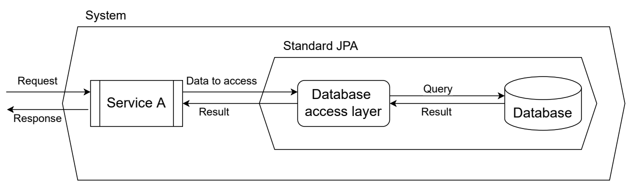

Fig. 1: General overview of system architecture.

But, GraphQL suffers from the N+1 problem [2] which can easily flood the database with unexpected requests. To solve this issue, the open-source community has created the Join-Monster [3] library, which pre-processes the GraphQL query, and creates a select to JOIN all the relations and run the query using a single database request. The issue with this approach is that this can very easily lead to a Cartesian product issue [4], which could severely tank the application's performance.

Both of these issues can be avoided by intelligently deciding the types of queries to be used based on the definitions of the entities, and, separating them into different sub-jobs, each responsible for a different segment of the entire query. RQL achieves this through the usage of multiple open-source libraries. Avoiding a restriction to the existing implementation of GraphQL, we use Cossium's EntityGraph implementation [5], which allows the dynamic generation of queries, based on the requirements that are sent to the back-end service. Another key library is MapStruct [6] because it helps us dynamically map all the attributes without risking automatic entity fetching while returning the data.

Data-parallel programming is a great paradigm that eases the parallelization of tasks and their implementation [7] which is used in RQL to map out the retrieved data to their corresponding parents. A similar idea to ours has been implemented on graph-based databases, which rewrites the queries into sub-queries, while also implementing alternate algorithms for mapping out and retrieving the data [8].

III. Database Access Optimizations and ORMs

Databases are powerful tools, but, problems arise when accessing them. Using the wrong approach for the wrong situation can mean extra seconds, minutes, or even hours during a simple lookup, in a "do anything" and "anyway" approach. Practically, they don't impose restrictions and protocols (unless they are technologically infeasible).

ORMs work in the same manner, they go by the KISS and SRP principles. They only take up the responsibility of giving database access in programming languages through objects, nothing more, nothing less. They don't try to optimize the user's database calls or try to impose any smart restrictions. This approach poses a problem because developers can make mistakes much more frequently. Two frequent types of problems due to ORMs using these principles are the N+1 issues and creating Cartesian products when they aren't wanted. Besides the principles, there is also the issue of "expected" undocumented behavior with ORMs. Such an example can be found in Spring Data JPA's "optional" property. More specifically, when an annotation has the "optional" property set to false, JPA **always** tries to access the records relating to that relation. It doesn't respect any other properties (like setting the fetch strategy as lazy), which can lead to unexpected behavior.

The new library helps to avoid this by imposing some restrictions, adding "patches" to the undocumented behavior, and making smart decisions about what actually should be accessed or not, while also giving the possibility to implement parallelization into the database access layer.

IV. Algorithms

A. General Overview

Usually, with most back-end frameworks, like Spring, the codebase is layered in such a way that there is at least a service layer that contains the business logic, and a data access layer (which is usually either an ORM or a lot of custom repositories that contain stored procedures or custom queries). In Fig. 1 we can see an example of this standard approach where Service A requests some data from Spring Data JPA (which in turn usually uses Hibernate entity manager as the data access layer) and then returns it as a result.

Fig. 3 presents a drawn-out example of this approach. Service A requests some data, but now, instead of directly calling the ORM, it first calls a divider (which also serves as an optimizer) to rebuild the query so that it runs efficiently, then, separates it into multiple sub-queries based on different criteria that are discussed later on in this same section. After that separation, it sends each of the sub-queries into different processing units that then call JPA. Afterward, when each processing unit receives the results (the parent entities as well as their relations), it sends it over to a local combiner so that any children can get re-mapped back to their parents which then gets returned as the result of each processing unit. The thread combiner collects these results and then combines them into one aggregate cohesive object that counts as the final result. Before returning the result from the RQL layer, an extra step is done which just wraps or unwraps the aggregate object so that the service layer can receive the result as if it had called JPA directly.

B. Setup

We use a specifically designed database that should break every undocumented behavior in Spring Data JPA (or in other words, designed to be as inefficient as possible), as presented in Fig. 2.

We've configured each X-to-one relation as a mandatory relation (which makes it so that even if we set the fetch type to lazy, JPA will still attempt to access these objects even if they aren't used later), for the accounts we have a many-to-many relation with itself which can easily

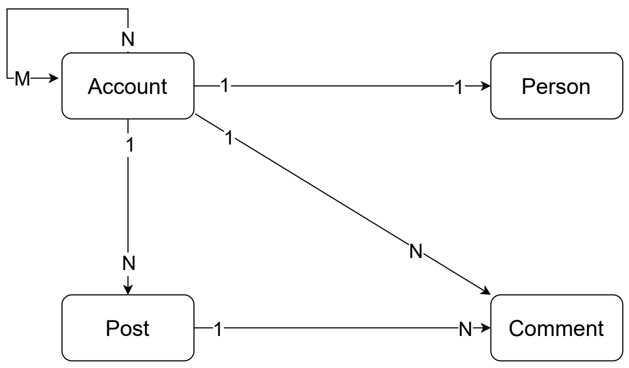

Fig. 2: ER diagrams of our sample database

throttle the performance of the application due to how JPA accesses all the relations. Another important part of the configuration is that we have defined the relation on both ends, for example in the Account and Person relation we have 2 properties, one which references the multiple posts, and one which references the accounts associated to them, which can also lead to a big throttle on performance and memory intensity due to the possibility of infinite recursion and possible depth. On a couple of the relations we also defined their "mappedBy" property which can also lead to JPA sometimes fetching the relation, even though it isn't requested or used. One more point that we'd like to stress is the circular reference between Account, Post and Comment (also keep in mind that all of these relations are 2 sided) which can lead to a heap of trouble in all aspects, performance, implementation, memory, complexity, etc. The configuration is designed in such a way that we had to override the equals and hash code methods because otherwise whenever we'd try to access any of the objects or methods, the application would simply crash.

C. Synchronous

First, we start with the synchronous aspect and optimizations of our algorithm.

Our starting point is a graph that RQL creates and uses to decide when to query specific entities. We call this graph a "reference graph". Whenever access to the database is requested, the first thing the library does is create a reference graph, which is essentially an ER diagram of the requested data with small modifications. One of those small modifications is the fact that whenever a cycle is encountered in the graph, it treats the "next in line" node as a separate node. The graph then adds the requested nodes to the tree, while also checking for bad relationship configurations (ex. a property is set with a lazy fetch strategy while also being a mandatory field). Based on the checks, it adds them in a specific way to the reference graph to avoid creating any n+1 issues by querying for them with a JOIN in their corresponding parent's batch select.

After creating the initial graph, the graph is then partitioned into special sub-graphs, based on the type of relationship in a BFS manner. If the relationship is an X-To-Many type, then it qualifies for a sub-graph. Then, it is also recursively invoked on those sub-graphs as well. While this is being processed, each sub-graph is marked with a depth level, which signifies their dependence on any previous sub-graphs. The main reason for this partitioning is to avoid Cartesian products and to add the capability for parallelism by separating one "giant" call into separate, discrete database calls.

Finally, after the partitioning, a batch query is run for each sub-graph recursively based on their depth level (each sub-graph waits for its parent so that they can receive the data to query). The query is generated through the usage of a Cossium's Entity Graph.

After all the data has been queried, it gets separated with a *HashMap* where each pair is made of the parent id as the key and a list of all its children as the value. After this separation, each parent gets their appropriate children set through reflection. Then, as the recursive functions return, the discrete data segments get combined into a final, cohesive result.

D. Parallelization

In terms of parallelism, the algorithm is implemented in such a way that it opens up multiple parallelization opportunities.

One successful attempt at parallelization was the following idea: Segment the original database call into discrete, smaller database calls where we try to gather the data as pages. This idea worked wonders. It gave the library a huge performance boost, making the calls $\sim$2-3 times faster. We implement this feature in 2 ways. The first one is by adding the capability to specify the number of threads for a particular call, and then divide the original request into that many pages. The other implementation is the inverse of the previous one, where the user would specify the maximum size of a page, and then the library would spin up as many threads as needed. With these approaches, the user can customize based on the resources they have available.

There is one downfall currently to this approach, which is a high possibility of "clogging up" the database connections since each page uses a separate connection.

Another successful attempt at parallelization was the idea of putting the HashMap segmentation and child mapping in separate threads, based on the count of child records (which can be set in spring's yml or properties configuration file). We make one big difference in the child mapping part, where we set an empty array if the property is null, and then we add the children to the list (rather than just outright setting the children). This didn't give us much performance, only about an 8-10% increase.

E. Remarks and Concerns

One point that we'd like to address is data leaks (ex. accessing a password of a user). With the algorithm model discussed thus far, this type of security is a great concern.

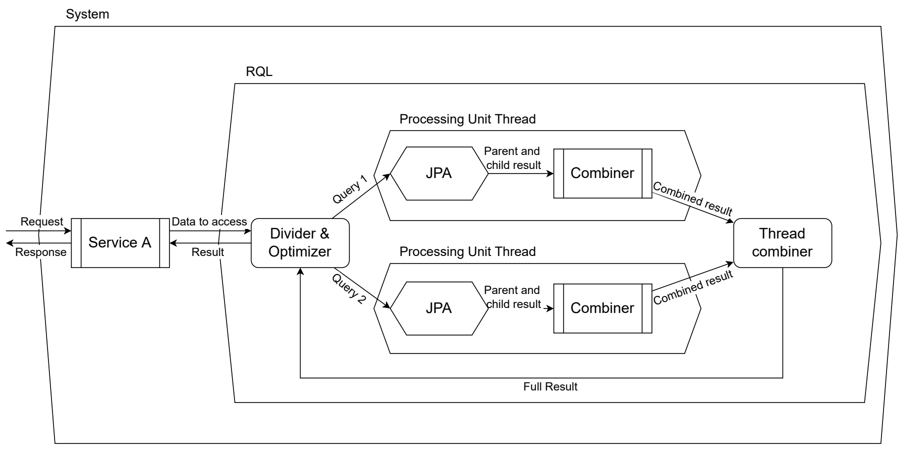

Fig. 3: General overview of proposed architecture using RQL.

We tackle this issue by adding the possibility to set an annotation on entity properties that are restricted. Then, through the usage of Spring AOP, RQL checks each database request for whether a restricted property has been requested and throws an adequate error.

Besides the possibility of data leaks, there is also the issue of over-fetching data. RQL also implements auto-generated MapStruct mappers which map the data to DTOs based on the reference graph. It can also be customized more specifically based on the scenario through the extension of those mappers, and the application of an *@InheritConfiguration* annotation [9].

V. Experimental Proof of Concept

A. Experimental methods

There are many scenarios and cases in which RQL is more or less optimized compared to its "competitors", and in this paper, we define 3 scenarios as the most important ones, each differing in the number of relations being requested, types of relations, and relational depth.

Scenario A is based on sending requests for accounts, together with all of their corresponding posts and comments. This scenario is meant to test how these libraries handle the fetching of multiple collections, which are also a part of another collection.

Scenario B will just be a simpler variant of Scenario A aiming at fetching accounts, and this time, only their posts. This is a very important scenario, precisely because it's so simple, and we'd expect that it should already be optimized.

Scenario C is dedicated to testing the depth use case, more precisely, focusing on taking accounts, their "person" relation, the account related to that person (essentially the original record), and then finally getting that account's "person".

The general basis for the testing is based on conducting tests and calculating the average of 20 requests per each number of records per scenario. The number of records per scenario is the same throughout the entire testing which is in sets of 50 records, more specifically, 50, 100, 150, 200, and 250 account records. Of course, for each account record, there will be another set of records that are fetched (but those amounts are discussed further on in this segment).

Furthermore, the database is set up in the following manner: for each account, we create 1 person and 100 posts, and then for each post, we create another 60 comments which we randomly assign to random accounts. On each initialization, we create 100 accounts, and for this test, 250 accounts are initialized (and their appropriate number of posts and comments). One more important point is that a timeout of 2 minutes is set for all cases. If a request hits this threshold, we classify it as if the result runs infinitely.

Since RQL also implements parallelization, we should also disclose the corresponding configuration. Our experiments use 5 threads, which get an equal amount of work given to them based on the algorithm.

The GraphQL queries have been defined as simple as possible since we're trying to encapsulate the performance based on the "ease of implementation" as well, or in other words, the amount of code and hours that would be needed for implementing GraphQL into a project should equal the amount it would take to implement RQL as well. The payloads in question also force the response data to be sorted based on the username of the accounts that are requested, so we can ensure that the data which is being sent as a response is the same.

B. Results

In scenario A the semi-standard JPA repository approach couldn't even operate, and while GraphQL can execute the experiment, it hits a soft cap at around 150 accounts, where the performance goes into the 2-minute mark. Other than that, RQL with our parallelization approach performs a lot faster in this case compared to the synchronous RQL approach. The semi-standard JPA failed because it created a Cartesian product that took over all

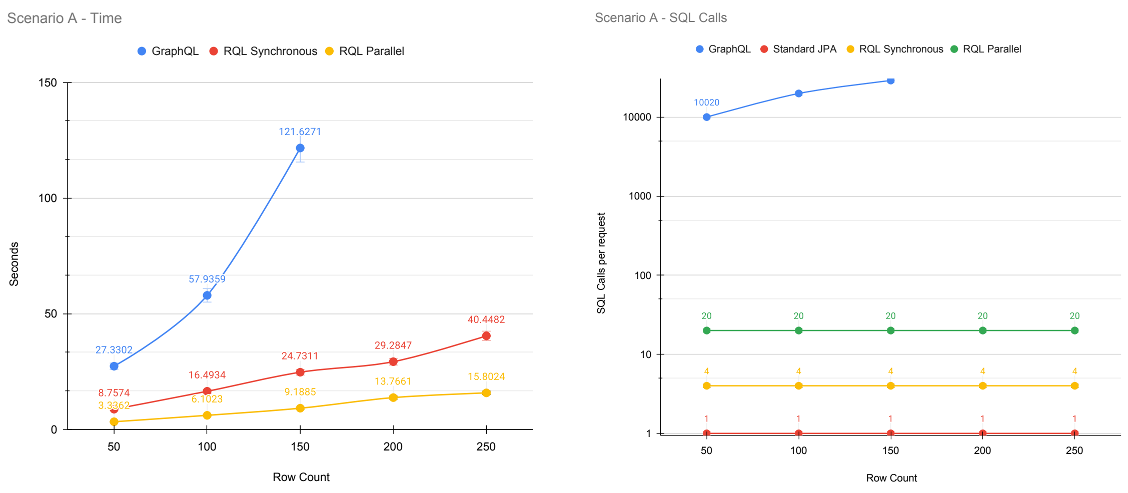

Fig. 4: Results of Scenario A.

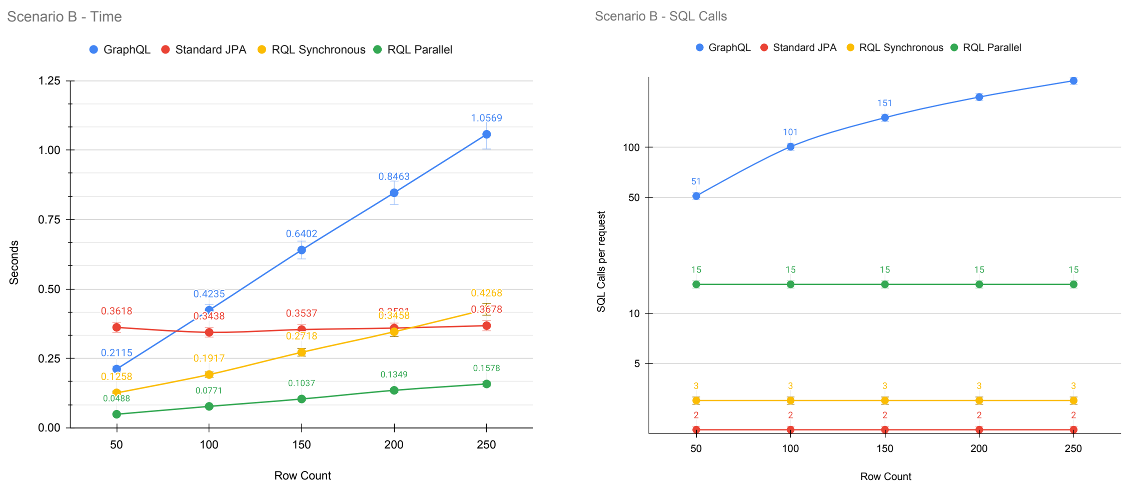

Fig. 5: Results of Scenario B.

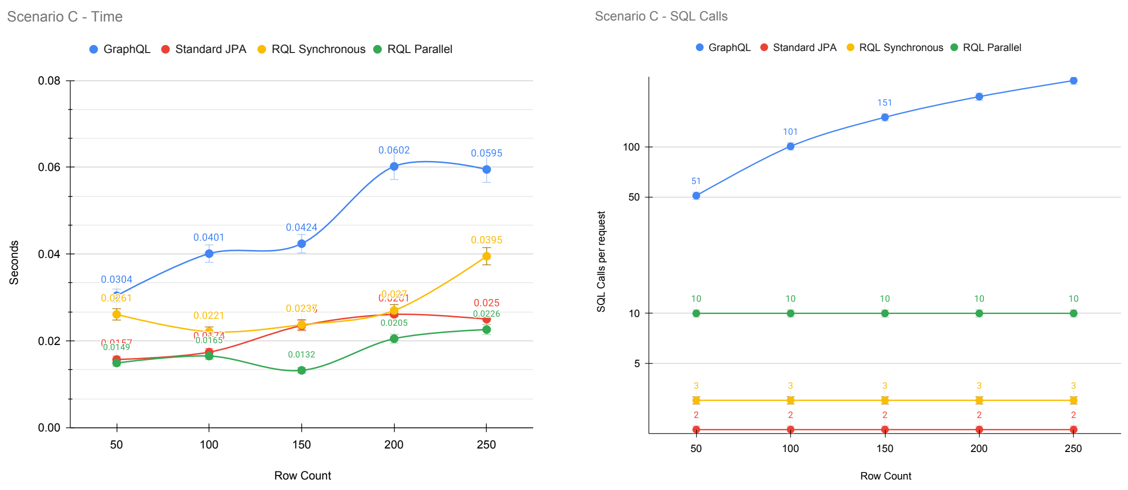

Fig. 6: Results of Scenario C.

of the system memory and hard disk memory which is dedicated to disk pagination (Fig. 4).

For scenario B the semi-standard JPA implementation managed to get to second place towards the end, and that's because RQL also values the system's memory, while JPA does not. More precisely, when running the semi-standard JPA implementation, it constantly kept taking more than the system memory made available to it (Fig. 5).

And finally, the interesting part for scenario C is the fact that the semi-standard JPA implementation was in second place (Fig. 6) throughout the entire testing.

C. Discussion

The parallelized version of RQL outperforms the synchronous one in terms of speed, while the synchronous approach uses fewer SQL calls. Other than that, both RQL approaches always outperform the GraphQL and semi-standard JPA implementations in both the number of SQL calls & speed.

Evaluating all achieved results, we conclude that indeed ORMs can improve a lot, confirming our hypothesis.

Another important fact that we can see from these results is that running a couple of extra SQL queries smartly can also drastically improve the performance and resource usage (since in the semi-standard JPA implementation we were having memory issues as discussed in Scenario A).

We also proved that some operations, currently aren't even executable with the existing technologies and their approaches.

The proof-of-concept example and insight into the introduced library are available at https://github.com/PegasusMKD/rest-graphql.

The existing implementation of the library does have a couple of edge cases, which are mainly regarding bad configurations, database & server connection throttling and deadlocks. For each of these issues, we can implement different solutions, for example, to avoid blocking server connections the library could implement a custom executor dedicated to it, and for the avoidance of database connection throttling, we can implement a producer-consumer pattern, where the library methods act as producers and we have a consumer that keeps said calls in a Queue and executes them. Finding the correct configuration has to be done with a trial-and-error approach and on a case-by-case basis. RQL will not inform the users at any point as to whether they've configured it properly or not since it can't guess the intentions, use cases, or machines that the code base will run on.

VI. Conclusion

Throughout this paper, we explained the algorithm, as well as the ideas and logic behind it. We also proved the hypothesis that by dividing the complete task into separate threads, proper decision-making, partitioning, in-memory data mapping, entity pre-processing, and the avoidance of Cartesian products and n+1 issues we can drastically improve any system's performance and resource requirements, as well as the strain it creates on the database itself.

We also discussed some issues and concerns with this algorithm like threads hoarding database connections or security issues, as well as ideas on how to fix them (ex. producer-consumer pattern for the hoarding of database connections). We've shown ways in which ORMs can improve and extend, and discussed some "hidden" behaviors within them. We also gave an alternative to GraphQL for companies which would provide some of its main features, while also drastically improving it in every aspect.

The next steps for our research would be making an official release of this library, improving on its stability, implementing a producer-consumer pattern for the parallel approach, testing it on real-world examples, and finally trying to re-implement it using different technologies and languages with more proper support for parallelization.

References

[1] The GraphQL Foundation. Graphql: A query language for your API. [Online]. Available: https://graphql.org/

[2] A. Ignjatovic. Understanding and fixing n+1 queries. [Online]. Available: https://medium.com/doctolib/understanding-and-fixing-n-1-query-30623109fe89

[3] Join-monster. [Online]. Available: https://join-monster.readthedocs.io/en/latest/

[4] A. Stankowski. (2019) Hibernate cartesian product problem. [Online]. Available: https://allaroundjava.com/hibernate-cartesian-product-problem/

[5] Cossium's entity graph. [Online]. Available: https://github.com/Cosium/spring-data-jpa-entity-graph

[6] MapStruct. Mapstruct: Java bean mappings, the easy way! [Online]. Available: https://mapstruct.org/

[7] C. Boyd, "Data-parallel computing: Data parallelism is a key concept in leveraging the power of today's manycore gpus." *Queue*, vol. 6, no. 2, pp. 30–39, 2008.

[8] F. Katsarou, N. Ntarmos, and P. Triantafillou, "Subgraph querying with parallel use of query rewritings and alternative algorithms," 2017. [Online]. Available: http://eprints.gla.ac.uk/130142/1/130142.pdf

[9] G. Morling, A. Gudian, S. Derksen, F. Hrisafov, and the MapStruct community. Inherit configuration documentation. [Online]. Available: https://mapstruct.org/documentation/stable/reference/html/#mapping-configuration-inheritance