Linköping University | Department of Computer and Information Science
Bachelor thesis, 16 ECTS | Datateknik
2018 | LIU-IDA/LITH-EX-G--18/058--SE

Result size calculation for Facebook's GraphQL query language

Beräkning av resultatstorlek för Facebooks GraphQL query language

Tim Andersson

Supervisor : Patrick Lambrix
Examiner : Olaf Hartig

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

GraphQL is a query language for web APIs and a service for carrying out server requests to interact with the data from the API. Research show that even for simple GraphQL queries both the size of the response object and the execution times to retrieve these objects may be prohibitively large and that current implementations of the language suffers from this issue. This thesis explores the implementation of an algorithm for calculating the exact size of the response object from a GraphQL query, and the performance based evaluation of the implementation. A proof of concept of a server using the implementation of the algorithm and subsequent tests of the calculation times for particularly problematic queries sent to the server, show that the implementation scales well and could serve as a way to stop these queries from executing.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

GraphQL is a query language for web APIs and a server-side runtime for executing queries using a type system that defines the data available.[8] This project consisted of implementing an algorithm for calculating the exact size of a response object from querying a GraphQL server, and to evaluate the performance of the implementation.

1.1 Motivation

In their research, Hartig and Perez [5] have shown formally that the GraphQL language emits highly efficient evaluation methods. However, they have also shown that the size of the response object from a GraphQL query can be exponentially large in terms of the size of the query. In such cases, producing and transferring the query result can be too resource intensive. To cope with this problem, they propose an algorithm for calculating the exact size of the response object from a query sent to a GraphQL server before the query is actually executed. The algorithm, however, is written in pseudo-code and based on a theoretical data model of a graph that is also defined in the article.

The main benefit of a working implementation of the algorithm would be the added control it gives to determine which queries the server should execute. With a working implementation, the server would be able to stop queries that are deemed too resource intensive to execute, which in turn would lead to a reduced server load. As the implementation would add a step to the GraphQL data retrieval process, it also comes with the price of adding some extra server load as well.

1.2 Aim

The purpose of this thesis project was to find a working implementation of the calculation algorithm and evaluate the performance of this implementation.

The main challenges of the work have been to find a way to keep the implementation of the algorithm not reliant on any specific back-end, and to implement a suitable test environment to test the full range of the calculation algorithm.

1.3 Research questions

1. How can Hartig and Pérez' calculation algorithm[5] be implemented in JavaScript and used in a GraphQL server?
2. How does the implementation affect the performance of the server response?

The criteria of a working implementation would be one that runs the algorithm after or during GraphQLs validation step but before the execution step. The implementation should be able to calculate the correct size of the result of a GraphQL query before execution and return this size if needed. The implementation should also be able to stop a query from executing if the size of the result is above a pre-set threshold. In this case, a GraphQL error should be raised. The performance of the server response will be measured in terms of the time it takes to calculate the result size of queries. Accesses to the database back-end will also be measured to analyze the performance of the test environment.

1.4 Delimitations

Hartig and Pérez[5] define a simpler syntax for GraphQL queries called non-redundant ground-typed normal form, which the algorithm also assumes the queries are in. The implementation will, for the sake of simplicity, be evaluated with non-redundant queries in ground-typed normal form.

2 Theory

2.1 GraphQL

GraphQL is a query language for web APIs and a service for carrying out server requests to interact with the data from the API. GraphQL uses a schema to describe the data available from the API and how the data relates to each other. Accompanying resolver functions for the schema determine how the data should be fetched from the database back-end.[7]

GraphQL schema

The GraphQL Schema defines what kind of objects can be fetched from the server and what properties, called fields, exist for these objects. The schema defines these objects in the form of a type system where a new type is added for every type of object. The fields are mapped by their name and what the field promises to return. The return values can be of scalar[1] types, other object types, or lists of these. Fields defined to return a list of objects or scalars are represented by adding square brackets around the field's return type. Fields can additionally have optional arguments and directives which the implementation of the schema can use to manipulate the returned objects.

```
type ProductFeature {
  nr: ID
  label: String
  comment: String
  products(limit: Int): [Product]
}
```

Listing 2.1: Example of an object type

An example of an object type in a GraphQL schema can be seen in Listing 2.1, this object type is used in the schema for the test environment of the project (see Section 3.2). The square brackets around the return type of the `products` field define that this field will return a *list* of objects of the `Product` type, which is another object type also defined in the schema. The `products` field also has an argument, `(limit: Int)`, which in this case a user can employ to specify a limit on the number of `Product` objects that will be returned. GraphQL also supports two kinds of abstract types called interfaces and unions. An interface defines

[1] Objects without sub-fields, concrete values like `Int`, `String`, `Boolean`, or custom types.

a set of fields and any object type that implements the interface is guaranteed to contain at least these fields. More fields specific to the object type can be added to the definition of the object type. A union defines a set of permitted types and if a type that returns a union type is queried, it is guaranteed that the results are of the permitted types. The schema must also define a special type called the `query` type. The fields in this type define the queries that serve as the entry points to the schema.

The schema is implemented in the form of resolver functions, which specify how the server should fetch the objects defined in the schema from the server back-end.[7]

GraphQL query

A GraphQL query is a string, in a JSON-like format, that describes what data is requested from the server. The GraphQL service running on the server executes the query and returns the expected data in JSON format back to the client. A simple GraphQL query and its response object can be seen in Listing 2.2 and Listing 2.3, respectively.

```
{
  Person(nr:6) {
    name
  }
}
```

Listing 2.2: GraphQL query

```
{
  "Person": {
    "name": "Caryn"
  }
}
```

Listing 2.3: Response object

The GraphQL query requests the object `Person` with an argument specifying that the person requested has the `nr:6`, and the field `name` of this object. The server returns the requested object and fields as key-value pairs. This results in the `Person` with `nr:6`, whose `name` is `Caryn`, in JSON format.[7]

GraphQL service process

The GraphQL service process for retrieving data via GraphQL queries performs two main steps for every request: validation and execution.

Validation

The validation step ensures that the given query conforms to the GraphQL schema of the server. It does so by testing the query through a set of pre-defined rules. GraphQL server frameworks usually have an option to add user-defined validation rules to this process. If any of the validation rules indicates an issue with the given query, the process raises a GraphQL error. In this case, the process forms a response object with the GraphQL error, stating where in the query the validation failed. If the validation raises any errors, the overall process will not continue to the execution step.[7]

Execution

If the query passes the validation step, it now can be executed. The appropriate resolver functions implementing the schema are called to retrieve the data requested. The process forms a response with key-value pairs in JSON format, where the key is the field queried and the value is the response. An extension field can also be present in the response map, where some middleware libraries can add information about the GraphQL process. If any field fails to be resolved(the resolver returned `null`, API was unavailable, etc.), a map called errors will be added to the response as well.[7]

2.2 The calculation algorithm

In the research article *Semantics and Complexity of GraphQL*[5] the authors prove that GraphQL queries can be written to return responses where both size and execution time are increasing exponentially by nesting queries. It is also shown through experiments that the current implementations of GraphQL suffer from this issue. The authors also provide a logical data model of a graph, as an abstraction of how GraphQL sees the underlying data of the backend; they call this a GraphQL graph. A GraphQL graph can informally be seen as a directed edge-labeled multigraph with a set of nodes, the nodes properties and edges between the nodes.

The issue of exponential increase by nesting queries becomes apparent when queries are written to form directed cycles with the list relationship between these nodes. If the nodes requested by a nodes list relationship cycle back to this node, the subquery to be executed for this node will be executed as many times as it is cycled back to. If this cycle continues back and forth, the times the same subquery is executed increases exponentially depending on the number of cycles. The authors do prove that this issue does not solely exist because of directed cycles and that queries with acyclic relations may also experience this exponential blowup. The problem lies in subqueries being executed for the same node multiple times. For the rest of the report, these kinds of queries will be referred to as cyclic and acyclic queries, respectively.

To cope with this problem, Hartig and Pérez[5] propose an algorithm in two parts, Algorithm 1, Algorithm 2, that can calculate the exact size of the response object for a GraphQL query in polynomial time, without having to produce the response object. The size of the response object in this context is the number of symbols in the response object. The calculation algorithm uses the notion of a GraphQL graph G. Algorithm 1 sets up two structures, labels and size. The labels structure is a map indexed by unique nodes and stores an array with the subqueries whose result size has been calculated for that node. This structure is kept to prevent the algorithm from calculating the same result size for the same node twice. This prevention is also what makes the algorithm be able to be run in polynomial time while the actual execution of the query may be exponential. The size structure keeps the calculated result size of every pair of nodes and queries that have recursively gone through the algorithm. The calculated result sizes of the subqueries are added to the parents of the nodes and the result size of the entire query will be available in size[r,φ], with r being a root node in the GraphQL graph, representing the Query type of the GraphQL schema (see Section 2.1). Algorithm 2 takes a GraphQL graph G, u which is a node in G, and φ which is the (sub)query over the node. The algorithm adds the number of symbols the response object will have to the size structure for every iteration. The algorithm can be broken into different conditions depending on how the subquery is structured:

- **Condition 1 - lines 3-8:** The (sub)query requests a single field, which will either have the response *field:value*, *field:[values]*, or *field:null* if the field does not exist on the node.
- **Condition 2 - lines 9-21:** The (sub)query requests a field with a subselection. The field is the relation the node has to the other nodes. For all the nodes that share this relation to the node, recursively run the algorithm for that node with the subselection as the query. For every node that shares the relation, also add 2 to the size structure to account for the curly brackets returned. If the type of the field is a list type, add 4 to the size structure to account for square brackets and *field:*, else if the field queried is not a list but rather one node with a subselection, add 2 to account for *field:*, else if the field is not a list type and no nodes sharing the relation can be found, add 3 to the size structure to account for *field:null*.

- **Condition 3 - lines 22-26:** The (sub)query requests a fragment[2]. If the type of the node matches the fragment type, run the algorithm for that node with the subselection of the fragment.
- **Condition 4 - lines 27-32:** The subquery consists of a set of multiple subqueries to be executed. Recursively run the algorithm for all the subqueries in the set.

Algorithm 1 `GraphQLSize`(G, φ), copied from [5]

Require: non-redundant φ in ground-typed normal form
1: **for all** node u in G **do**
2: $\texttt{labels}[u] := \varnothing$
3: **for all** ψ sub expression of φ **do**
4: $\texttt{size}[u, \psi] := 0$
5: **end for**
6: **end for**
7: Let r be the root node in G
8: $\texttt{Label}(G, r, \varphi)$
9: **return** $\texttt{size}[r, \varphi]$

Algorithm 2 `Label`(G, u, φ), copied from [5]

1: **if** $\varphi \notin \texttt{labels}[u]$ **then**
2: $\texttt{labels}[u] := \texttt{labels}[u] \cup \{\varphi\}$
3: **if** $\varphi = \texttt{f}[\alpha]$ **or** $\varphi = \ell\colon \texttt{f}[\alpha]$ **then**
4: **if** $(u, \texttt{f}[\alpha]) \in \mathrm{dom}(\lambda)$ **then**
5: $\texttt{size}[u, \varphi] := |\lambda(u, \texttt{f}[\alpha])| + 2$
6: **else**
7: $\texttt{size}[u, \varphi] := 3$
8: **end if**
9: **else if** $\varphi = \texttt{f}[\alpha]\,\{\,\varphi'\,\}$ **or** $\varphi = \ell\colon \texttt{f}[\alpha]\,\{\,\varphi'\,\}$ **then**
10: $V := \{v \mid (u, \texttt{f}[\alpha], v) \in E\}$
11: **for all** v in V **do**
12: $\texttt{Label}(G, v, \varphi')$
13: $\texttt{size}[u, \varphi] \mathrel{+}= \texttt{size}[v, \varphi'] + 2$
14: **end for**
15: **if** $type_{\mathcal{S}}(f) \in L_T$ **then**
16: $\texttt{size}[u, \varphi] \mathrel{+}= 4$
17: **else if** $type_{\mathcal{S}}(f) \notin L_T$ **and** $|V| \neq 0$ **then**
18: $\texttt{size}[u, \varphi] \mathrel{+}= 2$
19: **else**
20: $\texttt{size}[u, \varphi] := 3$
21: **end if**
22: **else if** $\varphi = \texttt{ on t}\,\{\,\psi\,\}$ **then**
23: **if** $\tau(u) = \texttt{t}$ **then**
24: $\texttt{Label}(G, u, \psi)$
25: $\texttt{size}[u, \varphi] := \texttt{size}[u, \psi]$
26: **end if**
27: **else if** $\varphi = \varphi_1 \cdots \varphi_k$ **then**
28: **for all** $i \in \{1, \ldots, k\}$ **do**
29: $\texttt{Label}(G, u, \varphi_i)$
30: $\texttt{size}[u, \varphi] \mathrel{+}= \texttt{size}[u, \varphi_i]$
31: **end for**
32: **end if**
33: **end if**

[2] an implemented type or part of a union, mentioned in Section 2.1

Running the algorithm before a given query is executed, enables a GraphQL server to determine if the query should be executed, based on the calculated size of the response.

2.3 The Node.js ecosystem

This project used Node.js to run an application serving a local GraphQL server, for the test environment. Node.js is a command line tool and an asynchronous event driven JavaScript runtime. It is based on Google's runtime implementation, the V8 engine, also used by the browser Chrome.[21] The runtime can be used to execute JavaScript code as applications, with access to special Node.js methods from a low-level I/O API, and functions from packages installed for the application. The *Node package manager*, npm, can be used to install packages published locally, on the npm website, or in git repositories. Packages are the building blocks of Nodes.js applications, defining functionality for the purpose of not having to repeat work. Published packages have a file called `package.json` in their root directory which defines information about the package such as its name, description, environments it can run in, and which other packages it depends on. Dependency packages can be specified with a specific version number, origin or an approximation of a version number that will work. When the package manager attempts to install a package, the packages it is dependant on according to the `package.json` file will also be installed and put in an optimal directory tree depending on which other packages might also be dependant on those packages. A package that has been published to the npm website is called a module. The functionality of modules ranges from providing a full-featured web-stack to modules with only one purpose like `is-number`, an improvement of the JavaScript code `typeof value === 'number'`.[10] For the rest of the report, a Node.js module can be considered the same as a JavaScript library.

2.4 JavaScript promises

In order to asynchronously query the database back-end, the implementation (see Section 3.2) and test environment used in the project used the notion of promises.

A JavaScript engine executing code will do so single threaded, synchronous and sequential in order to keep the concurrency of the final outcome. JavaScript does have support for asynchronous operations however and executes these by putting callbacks for asynchronous operations on the queue of an event loop. As long as any code is still in the execution stack, the event loop will not start handling the queue. This means that if an asynchronous function is called that will resolve its result in a callback, any code in same scope as the function call will run before the callback as these are on the execution stack.[14] A promise in JavaScript works as a proxy for a result that may not be available when the promise is created. Promises are asynchronous operations that can only fail or succeed once. A promise can have one of the following three states:

- *fulfilled* - the promise successfully completed and the outcome is available.
- *rejected* - the promise failed to complete and the outcome in the form of a rejection error is available.
- *pending* - the outcome of the promise is not yet known.

When a promise resolves and a result is available, either a value being fulfilled or a reason for error from rejection, the event handlers associated with the promise will be called. The event handlers determine what to do with the result of the promise and let asynchronous methods return values like synchronous methods. The promise prototype has a `.then()` method for adding event handlers to the promise and any code put in this method will not be ran before a result is available. Chaining together multiple `.then()` methods makes these run sequentially. As a promise is an asynchronous operation, any code not dependant on the promise

output can continue to run while the promise is not yet settled. A `Promise.all()` method exists that takes as input an array of promises and does not return until all these promises complete. As these are asynchronous operations, the order in which they are executed can not be guaranteed.[13]

2.5 Proof of concept

The project used a proof of concept GraphQL server to show a use case of the implementation of the algorithm. A proof of concept is documented evidence that a potential product or service can be successful[19], by demonstrating that the idea is feasible. In software development, this demonstration can be done by developing a product internally rather than introducing it to the public, to verify that a certain idea can be achieved in development. The term "proof of concept" is often used interchangeably with the term "prototype", and while these are similar methods, there exists some clear differences. A proof of concept does not take the usability in a real world scenario into consideration, because the end goal is not to use the developed product. A prototype, however, is a first attempt at a working model that is real-world usable. A proof of concept shows that an idea *can* be developed and validates the technical feasibility, while a prototype shows *how* it will be developed with an unrefined attempt at a final product.[1] When the proof of concept has served its purpose, it can usually be dismissed completely to start development of the deliverable product.[17]

2.6 Performance testing

Performance testing is a form of non-functional testing to determine parameters of performance measures of a system. Such testing can be used to verify that a system meets specifications or acceptance criteria, or to compare different implementations that have the same purpose. Microsoft breaks down the core performance testing activities into the following steps:[11]

1. **Identify Test Environment**
   The first step is to determine the logical and physical architecture and available tools that will be used in the testing. The environment used should be representative of the actual environment used in production. The physical architecture consists of hardware, software, and network configurations. An important consideration is the amount and type of test data that should be used to test the performance, to be representative of the actual data used in production. "Executing tests against a 1GB database when the live deployment will be 50GB is completely unrealistic"[12].

2. **Identify Performance Acceptance Criteria**
   The second step is to determine which metrics should be used when evaluating the test results and the acceptance levels of these metrics. Common metrics used are response time, throughput, and resource utilization. This identification process should lead to specified goals and requirements of the system.

3. **Plan and Design Tests**
   The third step is to identify and generate the test data. How the tests should be performed and how to collect the performance metrics should also be established. The goal of tests of performance should be to identify and report the performance metrics so that they can be evaluated later.

4. **Configure Test Environment**
   The fourth step is to set up tools such as load generation and application monitoring.

It should be made sure that the test environment is set up in a way that resource monitoring is both possible and accurate. After the test environment has been configured, it should be ready for performance testing.

5. **Implement Test Design**
   The fifth step is to integrate the method of collecting performance metrics into the designed tests. This step glues together the designed tests with the test environment. Tests previously designed should be set up to be ready for execution in the test environment.

6. **Execute tests**
   In this step, everything is set up and the tests are ready for execution. The tests are executed in the test environment, and metrics of performance are collected. The execution should lead to test results that can be analyzed and compared.

7. **Analyze, Report and retest**
   In the last step, the test results are available. These test results should now be analyzed with regards to the collected performance measures and the specified acceptance criteria. If a criteria is set as a threshold for a minimum value, and this criteria is not met, re-prioritize the tests and retest. Report the analyzed results with the data available, using the right statistics and preferably visually.

2.7 Performance measuring in Node.js

This section provides a brief overview of the performance metrics and their measurement as used in the project.

Execution time

The execution time is defined as the time spent by the system to execute a task, where executing in this context means actively using the processor resources. Measuring execution time in Node.js applications is typically done by collecting and comparing timestamps. The following methods from the Node.js API and functions from JavaScript exists for this, with varying levels of accuracy:

- `Date.now()` returns a UNIX timestamp[3] in milliseconds, based on the system clock.[3]
- `process.hrtime()` returns a high-resolution real time with nanosecond accuracy. The time is relative to an arbitrary time in the past, and thus not subject to clock drift[4]. This function can take an optional parameter with a previous call to `process.hrtime()` as input, to calculate the difference between the timestamps.[3, 16]
- `performance.now()` returns a high resolution timestamp with sub-millisecond accuracy, based on the time elapsed since the start of the Node process.[15]

Memory usage

A Node.js application stores all its memory inside a resident set, which consists of the code segment, stack, and heap. The code segment is the part containing the JavaScript code that is being executed by the process. The stack contains value types with pointers referencing objects on the heap. The heap is a memory segment that stores objects, strings, and closures. A Node.js method from the API called `process.memoryUsage()` can be used to return an object containing information about the resident set size, total allocated size of the heap, and the actual memory used by the heap while executing.[3, 9]

[3] Time in milliseconds elapsed since 1 January 1970 00:00:00 UTC
[4] A phenomenon where clocks start to diverge as they count time at different rates

3 Method

This chapter covers the steps in which the project was approached and carried out to answer the specified research questions. The project consisted of three main parts: analysis, implementation, and evaluation. The majority of these parts have been done in this order; however, sometimes the work on different parts was carried out simultaneously.

3.1 Analysis

To answer the first research question, the approach of developing a Node.js module implementing the algorithm was chosen. The approach was chosen because of the sizeable Node.js ecosystem, especially around GraphQL, and a Node.js module was determined as a likely scenario of how a production-ready implementation of the algorithm could be used.

Before anything was implemented, an analysis was done to specify the requirements that would be put on the implementation. The analysis was done by carefully examining the source code of the *graphql-js* Node.js module and the GraphQL specification. This was done to get a better understanding of the GraphQL service process, and how the GraphQL query is represented as a JavaScript object. The analysis also specified where the calculation should take place. Since the GraphQL process is usually performed by a server library, the source code of the server must be altered, unless it has support for middleware features. The decision was made to use a server framework with support for middleware features, and pass the implementation of the algorithm as an added rule to the validation step in the GraphQL process, described in Section 2.1. Since the criteria for a working implementation was that it should be able to calculate the correct size of the result and return this if needed, a way to return this result size was to be specified. The decision to print the result size in the server console using the JavaScript function `console.log()` was made, as for testing purposes this was an easy way to debug and confirm that the algorithm calculated the correct result. The implementation should also be able to halt the execution of a query if the calculated result size is above a certain threshold, and a way to do this had to be specified. However, since the calculation was decided to be part of the validation step, such a way to halt the execution was already implemented in the form of how GraphQL handles a failed validation rule. The chosen approach of answering the second part of the first research question, proving how the implementation could be used in a GraphQL server, was a proof of concept of a GraphQL server using the implementation. As described in Section 2.5, this server was not developed

with usability in a real-world scenario in mind, but rather to show that the idea of a server using the implementation can be achieved. To answer the second research question, the analysis also covered the preparations of performance testing of the implementation. Following the methodology of performance testing described in Section 2.6, as a first step, the test environment was identified. The back-end for the test environment was chosen to be generated using the Berlin SPARQL Benchmark data generator[2]. This tool was used to generate scalable `.SQL` files that could easily be read by SQLite to generate a database file with populated tables. This back-end was suitable for testing because a GraphQL schema representing it could be written to explore all conditions of the algorithm, and it allowed for both cyclic and acyclic queries. The Node.js module *apollo-tracing* was chosen to test the execution times of the test queries, as it has an option to add tracing information to the response from the GraphQL server. The tracing information already supported the return of the duration of the execution step. The build was forked and support for returning the duration of the validation step was added. The library uses the `process.hrtime()` function described in Section 2.7, and thus the metrics collected would be measured in nanoseconds. Tests were planned and designed by establishing which queries would be appropriate to test the implementation with. These queries were chosen to test specific scenarios in which the main benefits of the algorithm are either heavily- or under-utilized.

3.2 Implementation

The implementation of the algorithm and the proof of concept GraphQL server was worked on in parallel. To test the module being developed the GraphQL server was hosted locally, ran as a Node.js application.

Algorithm Implementation

The first implementation of the algorithm was very similar to how the algorithm is written in the pseudo-code but translated to JavaScript. The back-end and GraphQL server used a self-developed implementation of a small edge-labelled multi graph with globally unique node ids for the graph, as described by Hartig and Pérez[5]. The algorithm was first implemented and tested with queries over this small data set. Since the data set was very small, a full implementation of Algorithm 1 could be run for every node in the data set. The functions from Algorithm 2 for getting the edges for a node, the type of a node, and the size of a scalar object, were implemented in the graph implementation. The main challenge of the first implementation was to select a suitable data type to use for the size structure. The structure is indexed by both the node and query, [u,φ]. A natural choice for this data type would be a JavaScript Map with the object [u,φ] as the key and the size as the value. However, an initial attempt to use this was not successful as the JavaScript Map passes the keys by reference when objects are used, which means that two different objects used as keys in the Map can look exactly the same but still exist as separate keys. This becomes a problem when the algorithm tries to retrieve a value from the Map with a node and a subquery as the key; the object [u,φ_i] may look the same as the key requested but they are different instances of objects with the same properties and values. To solve this issue an implementation of a hashtable was done, which translates the keys to strings. This approach works because JavaScript does not pass strings by reference.

The implementation was determined as working or not based on the criteria specified in Section 1.3. To make sure that these criteria had been met, an in-browser IDE for exploring GraphQL, called GraphiQL[6] was used to query the server. Turning on GraphiQL is an option for most GraphQL server frameworks and the chosen framework for the test environment, *apollo-server-express*, has this feature as well. The GraphiQL interface can be seen in Figure 3.1. By using the GraphiQL interface, every field of every type was tested for some example entry points, by querying these fields and making sure that the response objects had

the correct values. The calculated response sizes were printed to the server console. The response sizes for small queries were calculated by hand and compared to the console prints to make sure that the size was calculated correctly. Some cyclic queries were used that requested the same field for the same node. Then, by checking debugging messages, it was made sure that upon entering the algorithm with these arguments, the algorithm would return with no extra calculations made.

The criteria of being able to stop a query from executing if the calculated result size is above a pre-set threshold was tested by passing a threshold to the algorithm. With the threshold, it was made sure that queries with result sizes below the threshold were allowed to execute, and queries with a result size above the threshold did not continue to the execution step. The GraphiQL interface was used to look at the response and make sure that a GraphQL error was returned when appropriate.

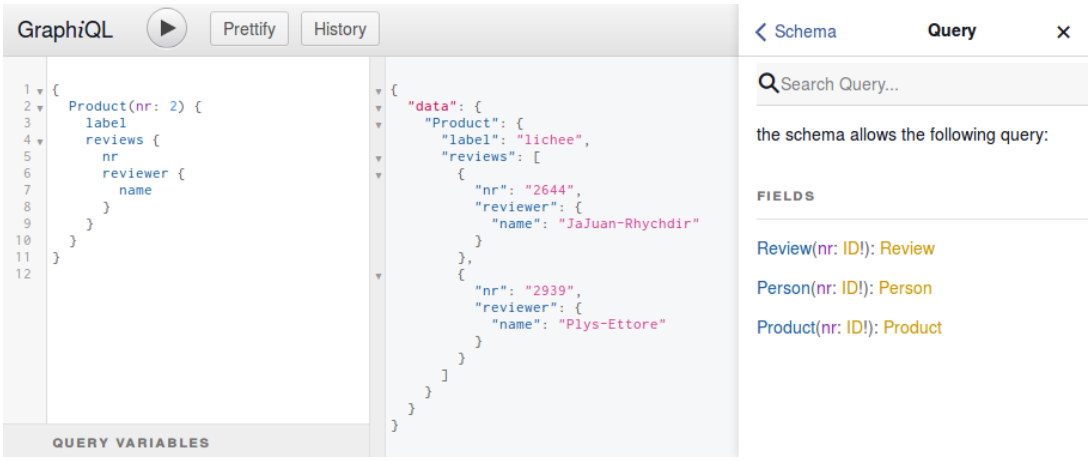


Figure 3.1: GraphiQL Interface. The query on the left, response in the middle, and schema exploration on the right.

After making sure that the algorithm worked with hard coded values, some alterations to the algorithm were done in preparation of using a relational database as the back-end. These alterations made all the back-end fetching functions promise-based, because of the asynchronous nature of querying a database. The algorithm was also split up into two parts, one part with the algorithm working as an interface that could stay the same for all implementations, and one part with the functions querying the back-end. This made the main algorithm part database-agnostic, while the functions are specific to the back-end used. To prepare for a larger data set, for which running lines 1-6 of Algorithm 1 with the entire data set would be inefficient, the decision to remove this part was made. With the part removed, lookups in the label and size hashtables were added when needed to initialize these. A back-end specific representation of the Node object was added, as the nodes in the new data set did not have globally unique ids to represent them.

GraphQL server

A small back-end using SQLite tables was set up. This new back-end was using the same data as in the hard coded version. The decision to use SQLite as the database for the back-end was done because of previous experience, ease of use, and excellent JavaScript support in the form of a library called *sqlite*. The alterations to the algorithm to make it promise-based proved to be a problem as the whole implementation now became asynchronous, which meant having to wait for all promises to resolve before determining if the result size is above the pre-set threshold or not. The actual reason for the problem turned out to be the decision to make the algorithm a part of the validation step, as this step is not designed to be asynchronous.

A decision was made to instead fork the build of the server module and add the calculation module as a dependency. The code was altered to have an extra step calling the calculation, between the validation and execution steps. The execution step was then put in the callback of the calculation to make sure it was not ran before the calculation was finished.

Test environment

By following the performance testing methodology (see Section 2.6), the test environment was configured by generating a new data set using the BSBM data set generator[2]. This was done to have a large enough data set for the testing. An ER-diagram of the data set that the tool generates can be seen in Figure 3.2. Properties for the entities are generated with random values using dictionaries for strings and different ranges for numbers.

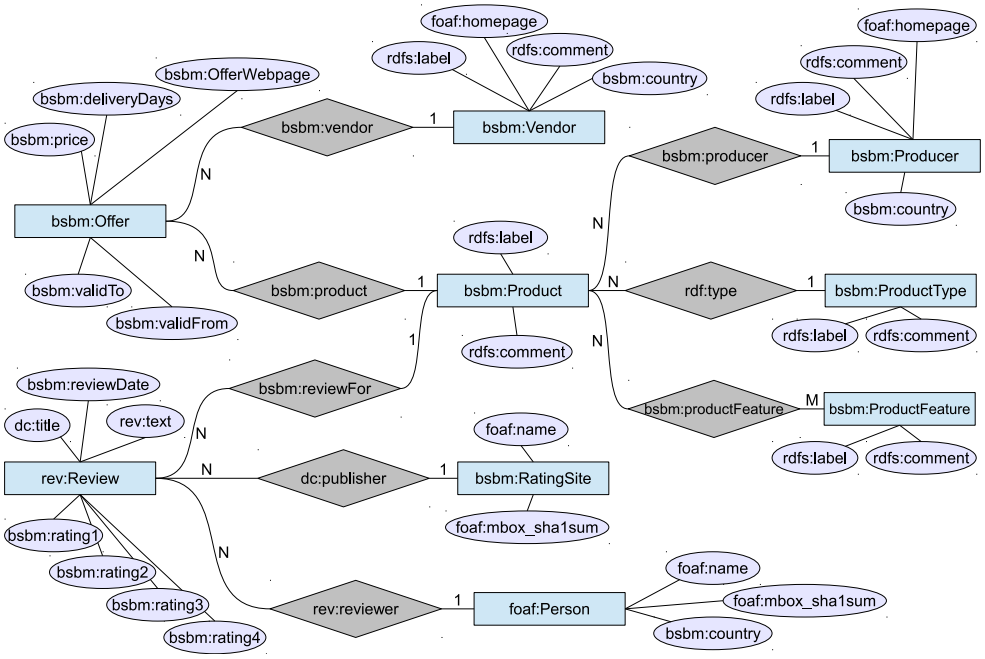


Figure 3.2: ER-diagram of the data set[4]

To generate the files for the data set, the `generate` shell script provided with the BSBM installation was run with the command `./generate -pc 666 -s sql`. This generated `.SQL` files with scripts for SQL database engines to create and to fill tables with data. These `.SQL` files are designed to be read by MySQL, and as SQLite lacks some functionality in comparison to MySQL, some modifications had to be done for SQLite to be able to read the files. A script for modifying the generated files was created and is available in Appendix B.1. The `-pc 666` flag of the generate command specifies the scale factor of the data set, and corresponds to the number of products to be created in the generated data set. An overview of the generated entities can be seen in Table 3.1. Entities also have a unique `nr` property as its index in the SQL tables.

Products	666
Producers	14
Product Features	2,860
Product Types	55
Vendors	8
Offers	13,320
Reviewers	339
Reviews	6,660
File size	16.2 MB

Table 3.1: Generated entities

The BSBM data set was extended to include one more relationship for the `Person` entity. This new relationship was called `knows` and the cardinality of this was a many-to-many relationship between `Person` entities. The relationship was added in the form of a new

table in the SQLite database and the `.SQL` file used for setting up this table can be seen in Appendix B.2. This new relationship was added to be able to test the implementation with acyclic queries. In order to do this, the table was filled with pairs of fixed values of the `nr` property from the `Person` table. A graph representation of the `knows` table can be seen in Figure 3.3, with every node representing a `Person` with its `nr` property inside the node. Note that the relationship forms a directed acyclic graph, and no `Person` knows another `Person` with a lower `nr` property than themselves. Every edge from the graph represents a row in the table, and a total of 31 `Person`s were affected by the relationship.

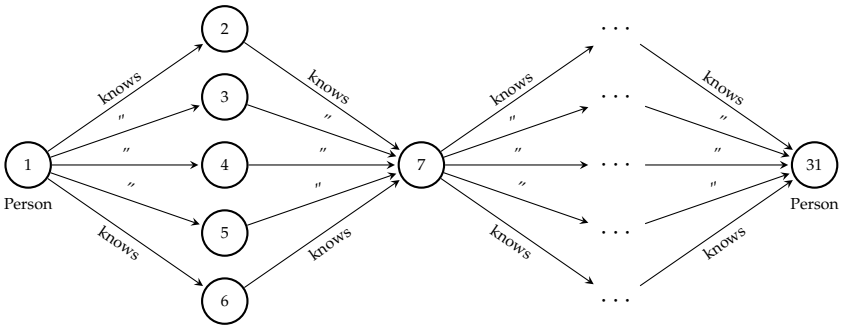


Figure 3.3: Graph representation of the table `knows`

A new GraphQL schema for this new data set was specified and made sure it was working properly. This was done by cross-referencing the values using the SQLite command-line interface to query the database back-end directly and comparing the values. The full schema used can be found in Appendix A.1. The GraphQL schema for this data set is defined with all the elements from the ER-diagram and the relationships appear in both directions for the entities, allowing cyclic queries. An example of the `Product` entity, represented as its own type in the schema, can be seen in Listing 3.1 along with the available queries over the data set.

```
type Product {
  nr: ID
  label: String
  comment: String
  producer: Producer
  type: ProductType
  productFeature: [ProductFeature]
  reviews: [Review]
  offers: [Offer]
}

type Query {
  Review(nr: ID!): Review
  Person(nr: ID!): Person
  Product(nr: ID!): Product
  Offer(nr: ID!): Offer
}
```

Listing 3.1: Extract from the GraphQL schema

3.3 Evaluation

To answer the second research question, the implementation was evaluated regarding its performance. The measurements evaluated were execution time and number of database accesses. The testing was done by executing a sequence of specified test queries in the test

environment, collecting the appropriate measures, to later analyze and report the result of the testing.

The test design was implemented by specifying these queries to apply to the BSBM-based test data set:

- **Q1 Cyclic Querying**
  α_0 = label , α_i = reviews{reviewFor { α_{i-1} }}, i > 0
  Query = Product(nr:6) { α_n }, n = Query Level

  The query retrieves the product with nr = 6. This in turn retrieves all reviews for this product, which in turn retrieves the product the review is for. This query will cycle back to the product with nr = 6, and this cycle will continue depending on the query level until it eventually requests the label field of the product with nr = 6. This query causes exponential result-size blowup, but will query the same node for the same field multiple times.

- **Q2 Acyclic Querying**
  α_0 = name , α_i = knows { α_{i-1} }, i > 0
  Query = Person(nr:1) { knows { α_n }}, n = Query Level

  The query retrieves the person with nr = 1. From here on all the other persons that this person knows are retrieved and this cycle continues depending on the query level until it eventually requests the name field of the last person(s). From Figure 3.3 it is clear that for Query Level 1 the name field of the person with nr = 7 will be requested, level 2 will request the name field of the persons with nr = 8-12, etc. Unlike the cyclic query, this query introduces new nodes for every query level.

- **Q3 Branching Query, varying fields**
  1 Field: Person(nr: 2) {**reviews** {**reviewFor** {**offers** {**vendor** {nr}}}}}
  2 Fields: Person(nr: 2) {name **reviews** {title **reviewFor** {label **offers** {price **vendor** {country nr}}}}}
  3 Fields: Person(nr: 2) {name country **reviews** {title reviewDate **reviewFor** {label comment **offers** {price deliveryDays **vendor** {country nr label}}}}}
  4 Fields: Person(nr: 2) {name country nr **reviews** {title reviewDate nr **reviewFor** {label comment nr **offers** {price deliveryDays nr **vendor** {country nr label homepage}}}}}

  Starting from the Person with nr = 2, query its relationships and their relationships ending up at vendors. Four different queries are used with varying number of fields queried on each level of the relationships. These queries do not have a cycle, and are used to test the added calculation time in a case where only a small number of fields are queried multiple times on the same node.

- **Q4 Extreme blowup, cyclic querying**
  α_0, β_0 = label, α_i = productFeature{ β_{i-1} }, β_i = product{ α_{i-1} } , i > 0
  Query = Product(nr:2) { α_n }, n = Query Level

  As the Product-ProductFeature is a many-to-many relationship, the result size will blow up fast with an increasing query level. The result-size algorithm will however visit the same nodes with the same queries multiple times, in which case no extra calculation is made.

- **Q5 Extreme blowup, with limit**
  α_0, β_0 = label, α_i = productFeature(limit:3){ β_{i-1} }, β_i = product(limit:3){ α_{i-1} } , i > 0
  Query = Product(nr:2) { α_n }, n = Query Level

  The same queries as in the extreme blowup case, but every level is limited by 3 nodes.

To test the execution time of the queries, a Node.js module was implemented as a test script and ran for every test query. This script can be found in Appendix B.3. The queries

used can be found in Appendix B.4. The execution time was measured by running the script ten times for every query. The script then generated a text file summarizing the average time it took for the result-size calculation and for the execution of the queries. Database accesses were measured by increasing an integer variable every time a query was sent to the database back-end. This measurement was taken because in a real life scenario of a GraphQL server the actual back-end may not be on the same machine as the server. If accessing the database is the bottleneck of the server, then database accesses should be limited. The measurement was taken both for the calculation and the execution step.

The memory that the server uses was planned to be measured using the Node.js method `process.memoryUsage()`, described in Section 2.7. This measurement was skipped when the metrics gathered looked like they were inconclusive to the same test runs. This was probably caused by the unpredictability of garbage collection time in Node.js.

As the execution time is highly dependant on the machine hosting the server, the results were ensured not to be misleading by doing all the testing on the same machine with the following specifications:

- Processor: AMD Ryzen 3 1200 3.4GHz 10MB
- Memory: 8 GB RAM
- OS: Windows 10 Education x64

Node.js v8.11.1 was used to host the server and run the test scripts. The versions of the most important Node.js modules used can be seen in Section 4.1.

4 Results

This chapter presents the result in how the algorithm was implemented, the proof of concept GraphQL server using the implementation, and the summarized results of the performance testing.

4.1 Implementation

Algorithm implementation

The resulting Node.js module implementing the algorithm can be installed with the npm[1] command line interface and is available as open source on GitHub. The module exports a function wrapping the algorithm called `queryCalculator`, which when run takes as arguments the back-end to be queried, the threshold for maximum result size and an object called `validationContext`. The `validationContext` object is the same object as the GraphQL validation step uses and it contains the query to be executed, along with the schema representing the back-end. If the result size turns out to be above the maximum threshold, the function adds a GraphQLError to the `validationContext` and returns the object. The idea is for the GraphQL server using the module to check the returned object for added errors and if there exists any, an error is returned in JSON format to the client and the server does not proceed to execute the query. This way of executing the function is exactly the same as the GraphQL validation step. The `queryCalculator` function is, however, promised-based like the execution step, and since JavaScript code is run asynchronously, the execution step must be chained together with the function to make sure the calculation is run before the execution step and not at the same time.

The `queryCalculator` initializes two hashtables, labels and sizeMap, that record the already calculated queries and the sizes of these queries, respectively. A function called `calculate` is the implementation of Hartig and Pérez' recursive calculation algorithm. The function has an extra argument passed to it called parent, which keeps track of the type of the parent to the current node in the algorithm. This is done solely for being able to execute lines 15-21 of Algorithm 2, as finding out if the node queried is a list type is done by introspection of the schema to see if the field on the parent is of the list type. For instance, in the query

[1] the default package manager for Node.js modules

Product(nr:2){producer{label}} the subquery for producer has a subselection but queries for only one node; the producer. Finding out if this relationship is a list type can not be done by looking at the query itself, but introspection of the schema marks the field on the `Product` type as "`producer: Producer`", which indicates that the relationship returns one object and therefore is not a list type.

The `queryCalculator` and `calculate` functions are both interfaced solutions meaning that they are not specific to any back-end. Back-end specific implementations of functions are passed to these functions to represent some of the conditions of the algorithm:

- `Node` is a constructor for a unique way to represent the node type from the back-end. A function for this type called `equals` also needs to be specified, for a way to compare equality of nodes when placed in the labels hashmap.
- `getRootNode` returns a unique node to the root node "Query".
- `getEdges` returns a list of unique nodes to recursively call the `calculate` function with. Corresponds to line 10 of Algorithm 2.
- `getNodeType` returns the type of a given node. Corresponds to $\tau(u)$ on line 23 of Algorithm 2.

How these functions get their data is up to the implementation of the functions. Support for querying the back-end is passed to the functions, but does not necessarily have to be used by the functions. For instance, in the implementation of these functions for the test environment used in the project only the `getEdges` function queries the back-end. Both the for loops on line 11 and 28 of Algorithm 2 ends up recursively calling the algorithm for every value in their respective collections. This is implemented with the JavaScript `promise-all()` function which returns a promise to asynchronously run the algorithm for all the values in these collections, meaning that different execution branches are created to switch back and forth when one is waiting. This is the desired behaviour because it does not matter which branch puts a query in labels, only that one and no more do. It's important to add references to the sizes of the subqueries and not their actual values as these might not have been calculated yet. This is because the algorithm adds the node and query to the labels hashtable as the first operation after determining that no other node has calculated it before. When this is added to the hashtable, it also prevents any other execution branch to calculate the query for that same node. The execution branch that put the query in labels may, however, end up waiting for a promise to resolve the `getEdges` function for example. While this branch is waiting, any other execution branch started at `promise.all()` can take over. If this new branch also ends up at the node and the query that were just put in labels, it will recognize that these are already put in labels and it will not calculate the size for it. The size will however not be available yet, because the first branch is still calculating it. For this reason, a reference to the object that will end up representing the size is added instead. The `queryCalculator` function waits for the entire calculation to resolve and then summarizes the values from the nested arrays at the hash for the root node, with the nested arrays being references to the sizes of the subqueries. The summarized value being the size of the response to the entire query is reported in the server console and compared to the given threshold.

Server libraries

Major relevant JavaScript libraries used for the implementation of the algorithm and test environment:

- graphql(v0.13.2), the JavaScript library was used. This library contains constructors for some types used. It also handles the GraphQL process of validation and execution.

- sqlite(v2.9.2) is a library for opening a connection and issuing queries to a SQLite database. The library was used because it is able to return queries wrapped in promises which allows for asynchronous calls to the database.
- graphql-tools(v3.0.1) was used to have an easier time writing the schema representing the back-end used in the test environment and map the resolver functions to this schema.
- apollo-tracing(v0.1.4) is a library for adding tracing information about the GraphQL process to the server response. Information about the duration of the entire query processing and individual resolver calls had already been supported in this library. The build was forked and support for the duration of the calculation step and the entire execution step were added.
- graphql-extensions(v0.0.10) is apollographql's middleware library for adding information to the optional extensions part of the GraphQL server response. The build was forked and functions for setting timestamps of the start and end of the calculation step were added.
- apollo-server-core(v1.3.6) is the library used by apollo-server-express to execute the GraphQL process. The build was forked and a new step to call the `queryCalculation` function before the execution, but after the validation, was added. Calls to the functions added to graphql-extensions were also added to measure the duration of the calculation and the execution of queries.
- jshashtable(v0.1.0) was used because the labels collection is using nodes as its indexes. The nodes in the implementation are JavaScript objects and trying to index a JavaScript object with another object is not possible. JavaScript interprets the object as a string [object Object]. Using a map structure was not possible either because using objects as keys passes the objects by reference. A Hashtable collection from jshashtable makes indexing the collection with objects possible.

GraphQL server using the implementation

As a proof of concept of the implementation working in a deployed GraphQL server, a modified version of the *apollo-server-core* module was used, as described in Section 4.1. Back-end specific functions were passed to the module. These functions can be found in Appendix A.3. In the SQLite database used as the back-end for the test environment, unique nodes are represented by a nr column that is unique to the table the node occurs in. Using this as the node representation function, the `Node` class has an id property storing the nr from the database and a table property storing the table in which the node is unique. The function comparing equality for nodes are comparing both the id and the table properties. The `getRootNode` function creates a new node with "id: 0" and "table: Query" and returns this node. The `nodeType` function passed just returns the table property, as the table names are the same as the types for the nodes in the database. The `getEdges` function is the only function querying the back-end. It is using directives set on the schema to determine how to build the SQLite `SELECT` query to select all nodes that share the relationship requested. Only the nr property of such a node is selected and an array containing new instances of the `Node` class is returned. The GraphQL server prints the result of calculated response objects in the server console and if any error is raised by the `queryCalculator` function, this error is returned to the client.

4.2 Evaluation

Execution time

The result for measured execution times for the queries specified in Section 3.3 are presented in order. The results are presented in the form of bar-charts, with the time taken by the calculation step as one bar and the time taken by the execution step as the other. The respective values of the times taken, in milliseconds, are also presented above all the bars. The result size of the response is also presented as an orange line with its y-axis to the right in the charts.

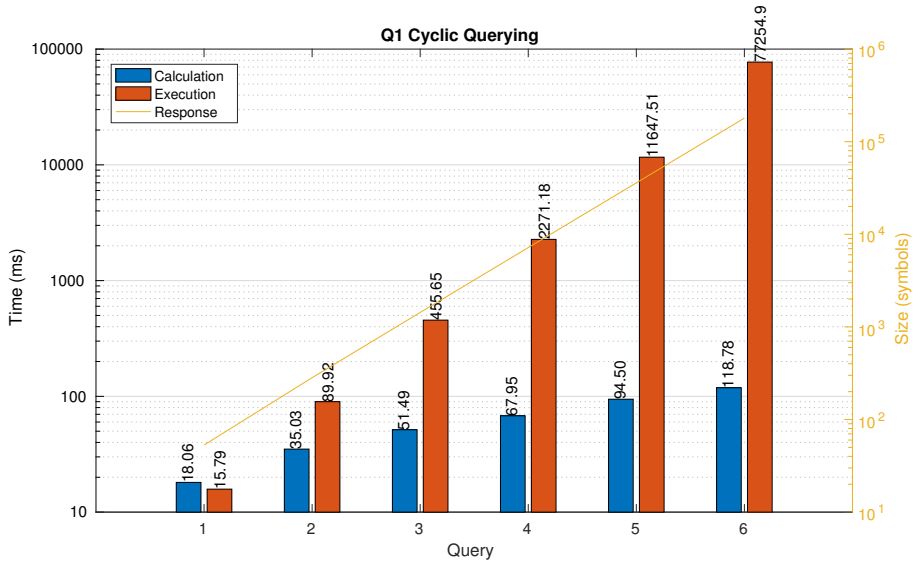


Figure 4.1: Cyclic Queries

The result of the first type of queries, Cyclic queries, can be seen in Figure 4.1. It should be noted that the y-axes are represented in a logarithmic scale, in order to show the result of both the steps. Both the execution time and the result size are increasing at a very rapid rate, while the time for the calculation algorithm starts out greater than the execution, to later only be a fraction (around 0.15%) compared to it for the sixth query.

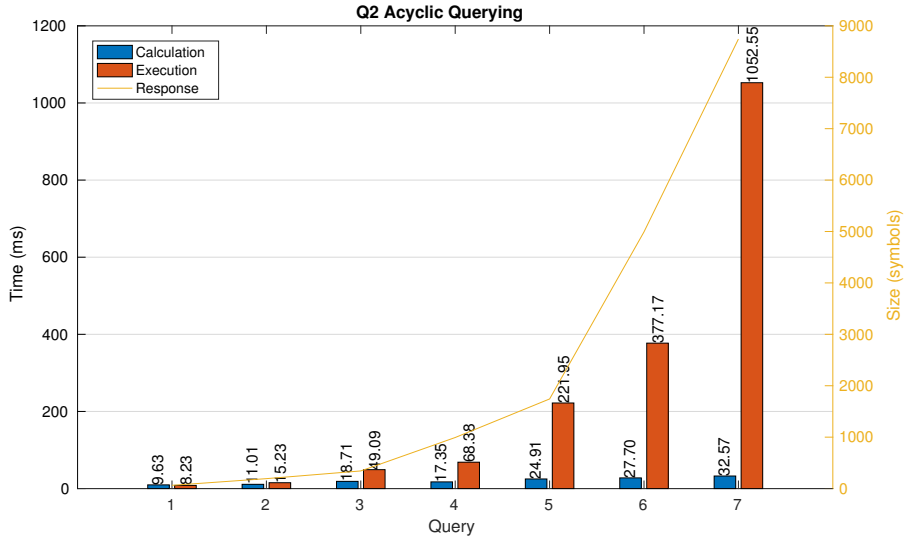


Figure 4.2: Acyclic Queries

The result of the second type of queries, Acyclic queries, can be seen in Figure 4.2. The result size and time taken for the execution step is clearly increasing exponentially, while the calculation time seems to have a more linear increase. The time taken for the calculation is greater than the execution for the first query.

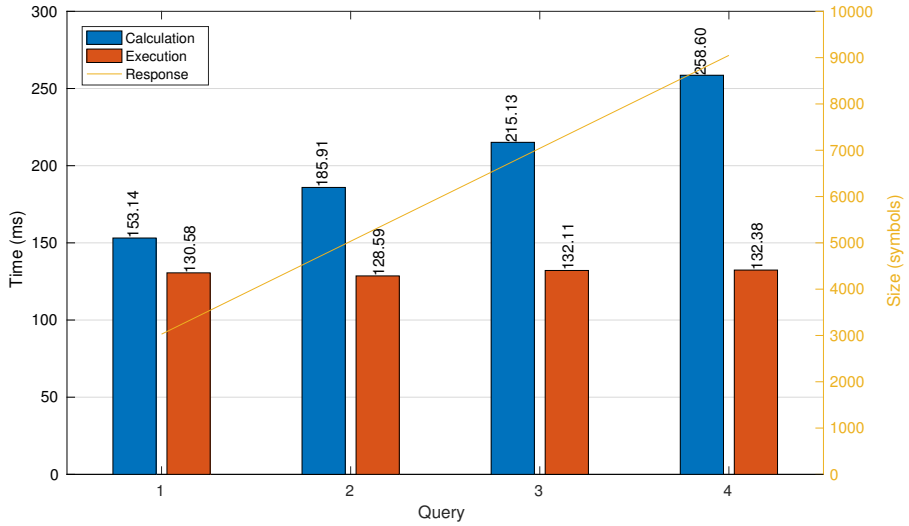


Figure 4.3: Queries with varying fields

The result of the third type of queries, Branching queries with varying fields, can be seen in Figure 4.3. Both the calculation time and response size increases linearly with more fields added. The execution time does not change with more fields added. In all of the queries the time for calculation is greater than that of the execution.

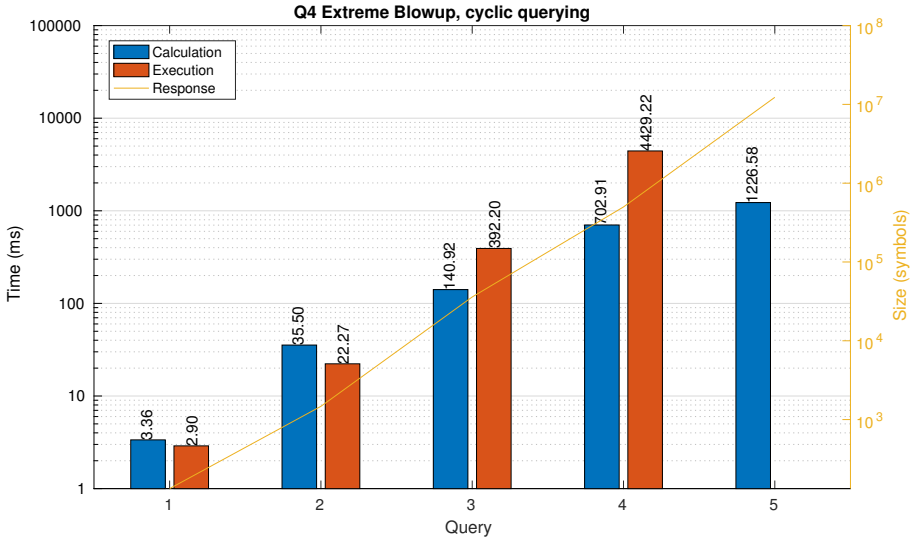


Figure 4.4: Queries with extreme blowup

The result of the fourth type of queries, Cyclic queries with extreme blowup, can be seen in Figure 4.4. It should be noted that the y-axes are represented in a logarithmic scale, in order to show the result of both the steps. For the fifth query, the execution step times out after around 170 seconds. The result size of the fifth query is 12,287,160 symbols.

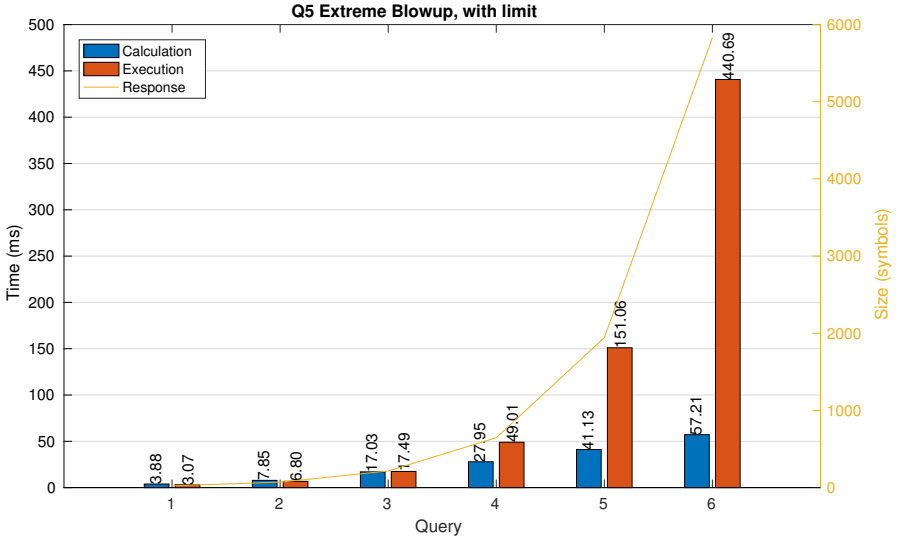


Figure 4.5: Queries with extreme blowup, limit: 3

The result of the fifth type of queries, Limited extreme blowup, can be seen in Figure 4.5. Since the list relations are limited to a cardinality of 3, the result size and the execution time clearly increase with a factor of 3 for increasing query levels. The calculation time increases almost linearly only.

Number of database accesses

The measured number of database accesses for the queries can be found in the tables below. The results are presented in the form of bar-charts, with the number of database accesses used by the calculation step as one bar and the same measure used by the execution step as the other. The respective values of the number of accesses are also presented above all the bars.

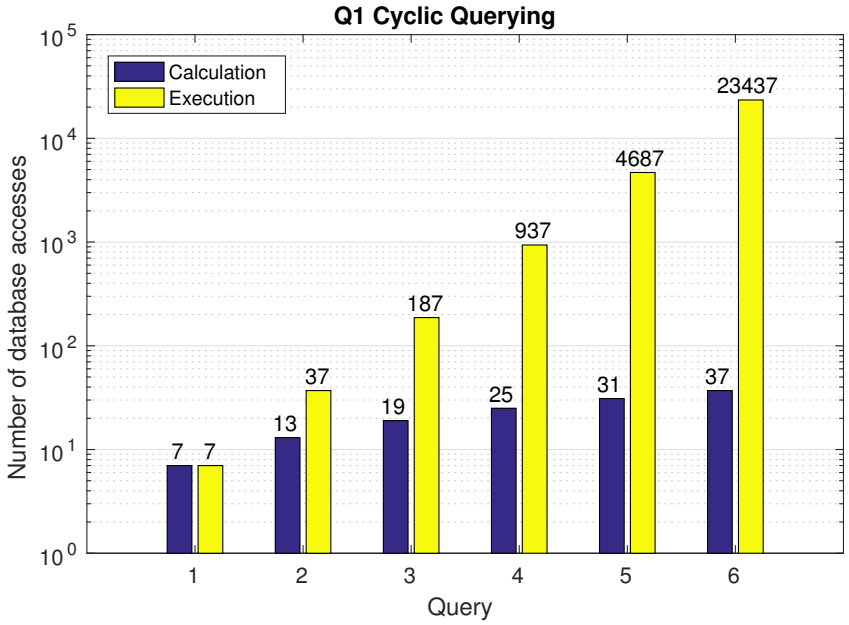


Figure 4.6: Cyclic Queries

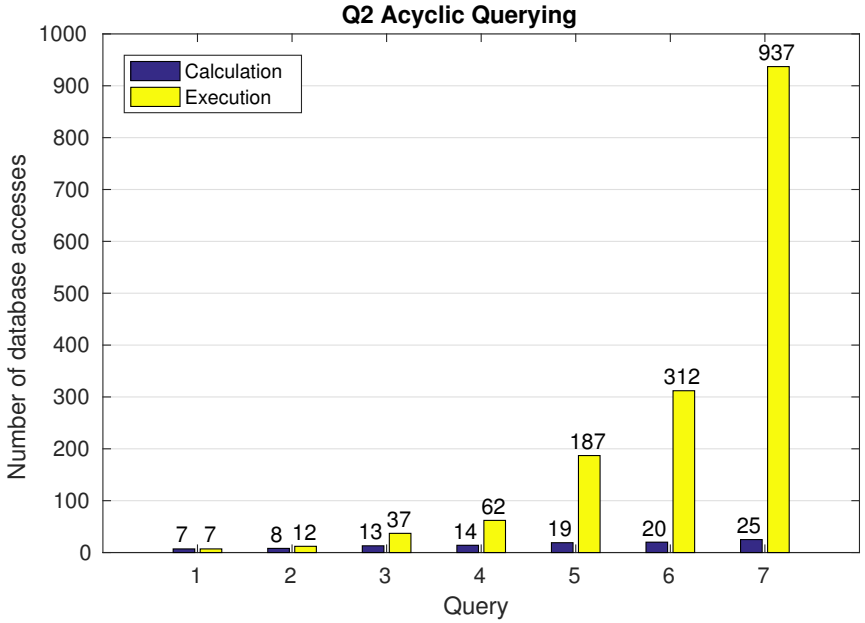


Figure 4.7: Acyclic Queries

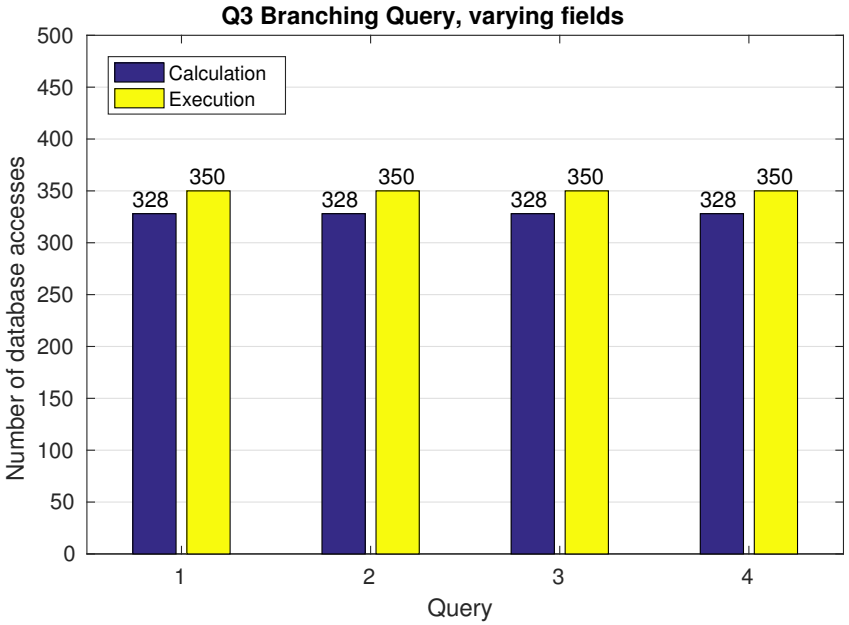


Figure 4.8: Queries with varying fields

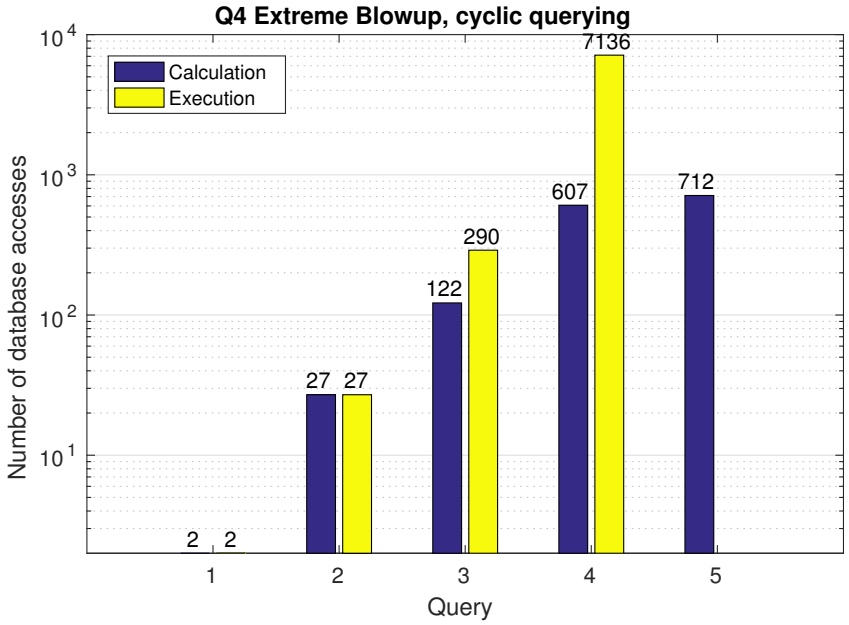


Figure 4.9: Queries with extreme blowup

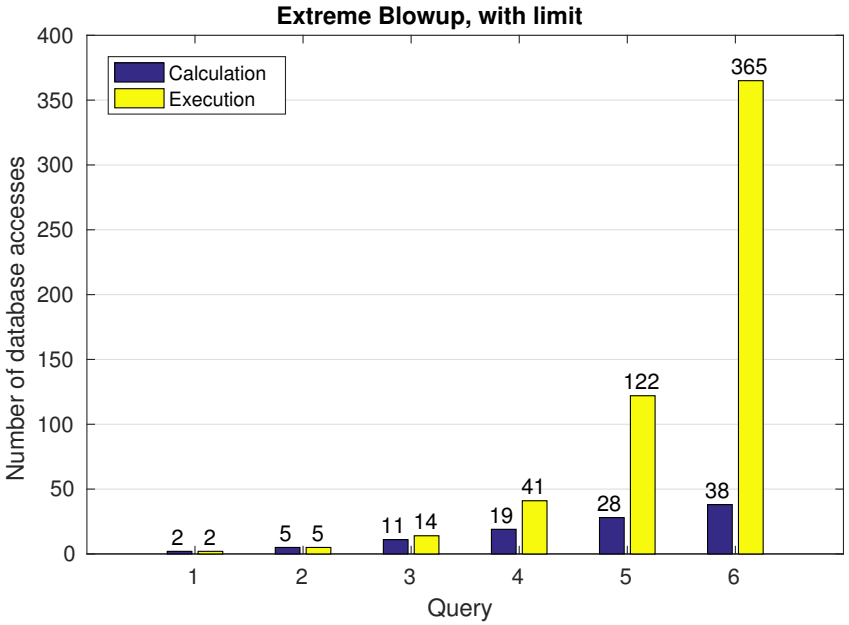


Figure 4.10: Queries with extreme blowup, limit 3

5 Discussion

5.1 Results

The time taken by the calculation and execution step for the queries with varying fields show that the time for the execution depends on how the resolvers function for the execution. The resolvers implemented for the test environment to query the back-end do so by selecting all the fields that exist for the object queried and then filtering depending on what fields are requested. This explains why the execution time stays the same when more fields are added, because these fields are already selected in all the queries, just filtered out. The results for the queries with varying fields and for the first query of all query sets, also show that in the case of queries without cycles, the calculation takes longer time than the execution. The reason for the calculation time being greater than the execution time in this scenario is believed to be the process of converting the subqueries and `Node` objects to strings for the hashtables. Because the query objects can be quite large, the time overhead of this "stringification" and comparisons of large strings for every lookup in the hashtables adds up to be greater than actually executing the query.

The results do however show that in the cases where the size of the response and the execution time increase exponentially with the query level, the calculation time does not. Hartig and Pérez[5] concluded that the algorithm is ran in polynomial time with $\mathcal{O}(|G| \cdot |\varphi|)$. As the algorithm was altered slightly in the implementation to not set up the labels and size hashtables in advance, and instead look up if nodes exists in these for every iteration of the algorithm, the implementation will have some time overhead for this. This will not impact the upper bound on the growth rate, because lookups in hashtables with "stringified" objects as the hash are in $\mathcal{O}(1)$.

The calculation does perform well compared to the execution time in the cases where the exponential blow up of the result size becomes a real problem. This can be seen in the sixth query of the cyclic queries (see Figure 4.1), where the time for the calculation step is around 0.15 seconds and the execution step is around 70 seconds. Another good example of this is the fifth query of the extreme blowup queries (See Figure 4.4), where the size of 12,287,160 symbols was calculated in about 1 second and the execution step timed out after 170 seconds.

The measured database accesses are in line with the measured execution times. The only database accesses the implementation carries out are when a list type is encountered on a node. From Figure 4.6 and Figure 4.7 it is clear that these values does not exponentially blow

up for the calculation step. This is because the list relations have already been calculated for the affected node. From Figure 4.3 and Figure 4.8 it can be established that the number of database accesses do not necessarily have to be a big factor in scaling the time taken for the calculation step, as an increase of around 66% can be seen from Query 1 to Query 4. It's also obvious from these results that the test environment suffers from the *n+1 requests* problem.[18] This happens when all the nodes fetched for a relationship send *one* `SELECT` query each for their subquery. In this case, batching the queries, sending one query for all the nodes and filter through these would be more efficient. This *n+1 requests* problem is a suboptimal way for a GraphQL server to function but can be solved using the *dataloader* Node.js module.

5.2 Method

The test environment served well in testing the implementation, however, the resolver functions to query the SQLite database back-end may have been too naively chosen. This is because sending a `SELECT` query for every resolver would perform poorly in a real scenario, as described in the previous section. These resolvers was, however, enough to test the algorithm implementation. Even though choosing the data structures to represent the label and size structures of Algorithm 1 was one of the more time consuming tasks in the project, even more consideration should have been put on this to increase the JavaScript code performance. As mentioned in the previous section, the choices for these data structures might be the reason for the calculation time being greater than the execution time in some cases.

The method of performance testing described in Section 2.6 has a lot of the focus on load- and stress-testing for full web applications. These measurements did not apply to this project, but the overall step by step methodology did, and was appropriately chosen. A method of performing performance testing in Node.js would be more specific to the implementation, but could not be found.

The condition of a query result being a sequence of scalars in Algorithm 2 was left out because the test environment did not have a good way to test this condition. This condition would easily be represented by a query of the list type but without a subselection(this is the way GraphQL represents a sequence of scalars) and a back-end specific function passed for retrieving the size of this set. Since the implementation is querying the database back-end with similar queries as the resolver functions uses, a way to cache the retrieved nodes to later use in the execution step would be an optimization for the server using the implementation. Another improvement of the implementation would be to use the resolver functions from the schema for the `getEdges` function (see Section 4.1) instead of having to write new functions for this. Resolvers for only querying for the unique identifiers of the tables would then probably have to be added to the schema, to avoid over-fetching.

Sources

The main source of reference used was *Semantics and Complexity of GraphQL*[5] which describes the algorithm the entire implementation is based on. The article is also the first research article to specifically focus on GraphQL. Given the relatively recent public release of GraphQL, not much information outside of blog posts and the official documentation for it was found. The information from blog posts without citations can certainly be questioned, but the official documentation and the research article was good sources used as a base of knowledge of GraphQL. A lot of information was gathered from documentations and source code of Node.js modules. The official documentation for Node.js is also released by the Node.js Foundation. Although the JavaScript documentation from MDN web docs used is a community effort, it is maintained by Mozilla and can thus be seen as a reliable source.

6 Conclusion

The research questions for this project were:

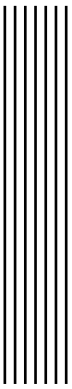
1. How can Hartig and Pérez' calculation algorithm[5] be implemented in JavaScript and used in a GraphQL server?

   A Node.js module was developed implementing the algorithm. To show a proof of concept of a GraphQL server using the algorithm, a test environment was implemented and a GraphQL server was hosted by a Node.js application. A step for calculating the response size was added to the GraphQL service process to run between the validation and execution steps. The implementation is able to calculate the correct size of the result of GraphQL queries, and print these to the server console. The added step to the GraphQL service process allows the server to halt the query before the execution step if the calculated size of the result is above a pre-set threshold.

2. How does the implementation affect the performance of the server response?

   The implementation was tested and evaluated by executing a sequence of specified test queries in the test environment, to later analyze and report the result of the testing. The metrics collected while testing were execution time and number of database accesses. The results of the execution time show that for simple queries without cycles, the time for the calculation step is slower than that of the execution step. However, the problem observed by Hartig and Pérez[5] was that queries can be written to exponentially increase in result size. Queries that have this exponential increase were tested and the execution time was noted to also increase exponentially, and thus getting out of hand quickly. The time of the calculation step for these queries was, however, noted to increase at most at a polynomial rate, and only be a fraction of the time of the execution step for deeply nested queries. The implementation can serve as an efficient way to stop queries deemed too resource intensive.

By answering the research questions my contribution is a working implementation of the calculation algorithm and a proof of concept GraphQL server using it. The implementation is an open source Node.js module that with some optimizations could be used for developers, to calculate the sizes of response objects for queries in GraphQL server implementations.



Bibliography

[1] Britt Armour. *A Beginner's Guide: POC vs. MVP vs. Prototype*. https://clearbridgemobile.com/beginners-guide-poc-vs-mvp-vs-prototype/. Blog post. 2017.

[2] Chris Bizer and Andreas Schultz. *Berlin SPARQL benchmark (BSBM) specification-v3.1*. 2011.

[3] Node.js Foundation. *Node.js v9.11.1 Documentation*. 2018. URL: https://nodejs.org/api/process.html.

[4] Olaf Hartig. "Querying a Web of Linked Data". PhD thesis. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, 2014. DOI: http://dx.doi.org/10.18452/17015.

[5] Olaf Hartig and Jorge Pérez. "Semantics and Complexity of GraphQL". In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1155–1164. ISBN: 978-1-4503-5639-8. DOI: 10.1145/3178876.3186014. URL: https://doi.org/10.1145/3178876.3186014.

[6] Facebook Inc. *GraphiQL*. 2015. URL: https://github.com/graphql/graphiql.

[7] Facebook Inc. *GraphQL Working Draft*. Oct. 2016. URL: http://facebook.github.io/graphql/October2016/.

[8] Facebook Inc. *Introduction to GraphQL*. 2018. URL: https://graphql.org/learn/.

[9] Daniel Khan. *Understanding Garbage Collection and hunting Memory Leaks in Node.js*. https://www.dynatrace.com/news/blog/understanding-garbage-collection-and-hunting-memory-leaks-in-node-js/. Blog post. 2015.

[10] Bevry Pty Ltd. *Node's Ecosystem*. URL: https://learn.bevry.me/node/ecosystem/ (visited on 06/20/2018).

[11] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. "Performance Testing Guidance for Web Applications: Patterns & Practices". In: *Microsoft press* (2007).

[12] Ian Molyneaux. *The art of application performance testing: Help for programmers and quality assurance*. " O'Reilly Media, Inc.", 2009.

[13] Mozilla. *JavaScript Reference*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[14] Mozilla. *JavaScript Reference*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Event_loop.

[15] Mozilla. *Performance Web API*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance.

[16] Tom Pawlak. *How to Measure Execution Time in Node.js*. https://blog.tompawlak.org/measure-execution-time-nodejs-javascript. Blog post. 2014.

[17] Odysseas Pentakalos. "Proof-of-Concept Design". In: *Skyscrapr (January 2008)< http://msdn. microsoft. com/en-us/library/cc168618. aspx* (2008).

[18] Arnaud Rinquin. *Avoiding n+1 requests in GraphQL, including within subscriptions*. https://medium.com/slite/avoiding-n-1-requests-in-graphql-including-within-subscriptions-f9d7867a257d. Blog post. 2017.

[19] Margaret Rouse. *Definition of proof of concept*. URL: https://searchcio.techtarget.com/definition/proof-of-concept-POC (visited on 06/28/2018).

[20] David B Stewart. "Measuring execution time and real-time performance". In: *Embedded Systems Conference (ESC)*. Vol. 141. 2001.

[21] Stefan Tilkov and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83.

A Appendix A

```
var {
  makeExecutableSchema
} = require('graphql-tools');

const typeDefs = `

type Vendor {
  nr: ID
  label: String
  comment: String
  homepage: String
  country: String
  publisher: Int
  publishDate: String
  offers: [Offer] @Offer(id: "nr", relation: "vendor")
}

type Offer {
  nr: ID
  price: Float
  validFrom: String
  validTo: String
  deliveryDays: Int
  offerWebpage: String
  product: Product @Offer(id: "product", relation: "nr")
  vendor: Vendor @Offer(id: "vendor", relation: "nr")
}

type Product {
  nr: ID
  label: String
  comment: String
  offers: [Offer] @Offer(id: "nr", relation: "product")
  producer: Producer @Product(id: "producer", relation: "nr")
  type: ProductType @producttypeproduct(id: "productType", relation: "product")
  productFeature(limit: Int): [ProductFeature] @Productfeatureproduct(id: "
      productFeature", relation: "product")
  reviews: [Review] @Review(id: "nr", relation: "product")
}

```

```
type ProductType {
  nr: ID
  label: String
  comment: String
  products: [Product] @producttypeproduct(id: "product", relation: "productType")
}

type ProductFeature {
  nr: ID
  label: String
  comment: String
  products(limit: Int): [Product] @Productfeatureproduct(id: "product", relation: "
      productFeature")
}

type Producer {
  nr: ID
  label: String
  comment: String
  homepage: String
  country: String
  products: [Product] @Product(id: "nr", relation: "producer")
}

type Review {
  nr: ID
  title: String
  text: String
  reviewDate: String
  rating1: Int
  rating2: Int
  rating3: Int
  rating4: Int
  reviewFor: Product @Review(id: "product", relation: "nr")
  reviewer: Person @Review(id: "person", relation: "nr")
}

type Person {
  nr: ID
  name: String
  mbox_sha1sum: String
  country: String
  reviews: [Review] @Review(id: "nr", relation: "person")
  knows: [Person] @Knowsperson(id: "friend", relation: "person")
}


  # the schema allows the following query:
  type Query {
    Review(nr: ID!): Review @Review(id: "nr", relation: "nr")
    Person(nr: ID!): Person @Person(id: "nr", relation: "nr")
    Product(nr: ID!): Product @Product(id: "nr", relation: "nr")
    Offer(nr: ID!): Offer @Offer(id: "nr", relation: "nr")
  }
`;

const resolvers = {
  Query: {
    Review: (_, {nr}, context)  =>
      context.db.get('SELECT nr,title,text,reviewDate,rating1,rating2,rating3,
          rating4,product,person FROM Review WHERE nr = $id', {$id: nr}),
    Person: (_, {nr}, context) =>
      context.db.get('SELECT nr,name,mbox_sha1sum,country FROM Person WHERE nr =
          $pid', {$pid: nr}),
    Product: (_, {nr}, context) =>
```

```
102       context.db.get('SELECT nr,label,comment,producer FROM Product WHERE nr =
              $pr_id', {$pr_id: nr}),
103     Offer: (_, {nr}, context) =>
104       context.db.get('SELECT nr,price,validFrom,validTo,deliveryDays,offerWebpage,
              vendor,product FROM Offer WHERE nr = $id', {$id: nr})
105   },
106   Vendor: {
107     offers(vendor, args, context){
108       return context.db.all('SELECT nr,price,validFrom,validTo,deliveryDays,
              offerWebpage,vendor,product FROM Offer WHERE vendor = $id', {$id: vendor.
              nr});
109     }
110   },
111   Offer: {
112     vendor(offer, args, context){
113       return context.db.get('SELECT nr,label,comment,homepage,country,publisher,
              publishDate FROM Vendor WHERE nr = $id', {$id: offer.vendor});
114     },
115     product(offer, args, context){
116       return context.db.get('SELECT nr,label,comment,producer FROM Product WHERE nr
              = $id', {$id: offer.product});
117     }
118   },
119   Review: {
120     reviewer(review, args, context) {
121       return context.db.get('SELECT nr,name,mbox_sha1sum,country FROM Person WHERE
              nr = $pid', {$pid: review.person});
122     },
123     reviewFor(review, args, context) {
124       return context.db.get('SELECT nr,label,comment,producer FROM Product WHERE nr
              = $pr_id', {$pr_id: review.product});
125     }
126   },
127   Person: {
128     reviews(person, args, context) {
129       return context.db.all('SELECT nr,title,text,reviewDate,rating1,rating2,rating3
              ,rating4,product,person FROM Review WHERE person = $pid', {$pid: person.nr
              });
130     },
131     knows(person, args, context){
132       return context.db.all('SELECT nr,name,mbox_sha1sum,country FROM Person p LEFT
              JOIN knowsperson kp on p.nr = kp.friend WHERE kp.person = $pid', {$pid:
              person.nr});
133     }
134   },
135   Product: {
136     producer(product, args, context){
137       return context.db.get('SELECT nr,label,comment,homepage,country FROM Producer
              WHERE nr = $pr_id', {$pr_id: product.producer});
138     },
139     type(product, args, context){
140       return context.db.get('SELECT nr,label,comment FROM producttype p LEFT JOIN
              producttypeproduct ptp on p.nr = ptp.productType WHERE ptp.product =
              $pr_id', {$pr_id: product.nr});
141     },
142     productFeature(product, {limit}, context){
143       if(limit){
144         return context.db.all('SELECT nr,label,comment FROM productfeature p LEFT
                JOIN productfeatureproduct pfp ON p.nr = pfp.productfeature WHERE pfp.
                product = $pr_id LIMIT $limit;', {$pr_id: product.nr, $limit: limit});
145       }
146       else{
147         return context.db.all('SELECT nr,label,comment FROM productfeature p LEFT
                JOIN productfeatureproduct pfp ON p.nr = pfp.productfeature WHERE pfp.
                product = $pr_id;', {$pr_id: product.nr});
148       }
```

```
149     },
150     reviews(product, args, context) {
151       return context.db.all('SELECT nr,title,text,reviewDate,rating1,rating2,rating3
            ,rating4,product,person FROM Review WHERE product = $pr_id', {$pr_id:
            product.nr});
152     },
153     offers(product, args, context){
154       return context.db.all('SELECT nr,price,validFrom,validTo,deliveryDays,
            offerWebpage,vendor,product FROM Offer WHERE product = $pr_id', {$pr_id:
            product.nr});
155     }
156   },
157   ProductFeature: {
158     products(feature, {limit}, context){
159       if (limit){
160         return context.db.all('SELECT nr,label,comment,producer FROM product p LEFT
              JOIN productfeatureproduct pfp ON p.nr = pfp.product WHERE pfp.
              productfeature = $fid LIMIT $limit;', {$fid: feature.nr, $limit: limit})
              ;
161       }
162       else{
163         return context.db.all('SELECT nr,label,comment,producer FROM product p LEFT
              JOIN productfeatureproduct pfp ON p.nr = pfp.product WHERE pfp.
              productfeature = $fid;', {$fid: feature.nr});
164       }
165     }
166   },
167   ProductType: {
168     products(type, args, context){
169       return context.db.all('SELECT nr,label,comment,producer FROM product p LEFT
            JOIN producttypeproduct ptp ON p.nr = ptp.product WHERE ptp.producttype =
            $tid;', {$tid: type.nr});
170     }
171   },
172   Producer: {
173     products(producer, args, context){
174       return context.db.all('SELECT nr,label,comment,producer FROM product WHERE
            producer = $pid;', {$pid: producer.nr});
175     }
176   }
177
178 };
179
180 const Schema = makeExecutableSchema({
181   typeDefs,
182   resolvers
183 });
184
185 module.exports = Schema;
```

Listing A.1: schema.js

```
var _ = require('lodash');
var Hashtable = require('jshashtable');
var HashTableArrays = require('./hashtable');
var deleteKey = require('key-del');
var {
  GraphQLError
} = require('graphql');
var {
  getEdges,
  getRootNode,
  nodeType
} = require('./functions');


function arrSum(arr) {
  return arr.reduce(function fn(a, b) {
    if (Array.isArray(b)) {
      return b.reduce(fn, a);
    } else if (b === Math.round(b)) {
      return a + b;
    }
    return a;
  }, 0);
}

const queryCalculator = (g, maxSize, validationContext) => {
  try {
    if (_.size(validationContext.getDocument().definitions) > 1) {
      return Promise.resolve(validationContext);
    }

    const calculate = (u, query, parent) => {

      if (!(_.some(labels.get(u), function(o) {
          return _.isEqual(o, query);
        }))) {
        if (!labels.containsKey(u)) {
          labels.put(u, [query]);
        } else {
          labels.get(u).push(query);
        }

        if (query.length > 1) {
          //  console.log("MultiQuery:C4");
          return Promise.all(query.map(item => {
            sizeMap.add([u, query], sizeMap.ret([u, [item]]));
            return calculate(u, [item], parent);
          }));

        } else if (!(query[0].selectionSet)) {
          //  console.log("FieldQuery:C1");
          sizeMap.add([u, query], 3);
          return Promise.resolve();

        } else if (query[0].kind === 'Field') {
          //  console.log('ListQuery:C2');
          let fieldDef = parent.getFields()[query[0].name.value];
          let currParent = fieldDef.astNode.type.kind === 'ListType' ? fieldDef.type
              .ofType : fieldDef.type;
          return getEdges(g, query[0], u, fieldDef)
            .then(result => {
              ++accesses;
              if (fieldDef.astNode.type.kind === 'ListType') {
                sizeMap.add([u, query], 4);
              } else if (result.length > 0) {
                sizeMap.add([u, query], 2);
```

```
              } else {
                sizeMap.add([u, query], 3);
              }
              return Promise.all(result.map(item => {
                sizeMap.add([u, query], 2);
                sizeMap.add([u, query], sizeMap.ret([item, query[0].selectionSet.
                    selections]));
                return calculate(item, query[0].selectionSet.selections, currParent)
                    ;
              }));
            });

        } else if (query[0].kind === 'InlineFragment') {
          //console.log('InlineFragmentQuery:C3');
          let onType = query[0].typeCondition.name.value;
          if (nodeType(g, u) === onType) {
            sizeMap.add([u, query], sizeMap.ret([u, query[0].selectionSet.selections
                ]));
            return calculate(u, query[0].selectionSet.selections, validationContext.
                getSchema().getType(onType));
          } else {
            return Promise.resolve();
          }
        }

      } else {
        //  console.log('query exists in labels');
        return Promise.resolve();
      }
    };

    var accesses = 0;
    var labels = new Hashtable();
    var sizeMap = new HashTableArrays();
    var query = deleteKey(validationContext.getDocument().definitions[0].
        selectionSet.selections, 'loc');
    let queryType = validationContext.getSchema().getQueryType();
    const rootNode = getRootNode(g, queryType);

    return calculate(rootNode, query, queryType)
      .then(() => {
        const querySize = arrSum(sizeMap.ret([rootNode, query]));
        console.log('Size of result: ' + querySize);
        console.log('Number of database accesses: ' + accesses);
        if (querySize > maxSize) {
          validationContext.reportError(
            new GraphQLError(
              `Validation: Size of query result is ${querySize}, which exceeds
                  maximum size of ${maxSize}`)
          );
        }
        return validationContext;
      });

  } catch (err) {
    {
      console.error(err);
      throw err;
    }
  }
};

module.exports = queryCalculator;
```

Listing A.2: calculate.js

```
function Node(x, y) {
  this.id = x;
  this.table = y;
}

Node.prototype.equals = function(obj) {
  return (obj instanceof Node) &&
    (obj.id === this.id) &&
    (obj.table === this.table);
};

var getEdges = (graph, query, u, fieldDef) => {
  const directives = fieldDef.astNode.directives[0];
  let table = directives.name.value;
  let u_id;
  let limit;
  if (query.arguments.length > 0) {
    if(query.arguments[0].name.value == 'limit'){
      limit = query.arguments[0].value.value;
      u_id = u.id;
    }
    else{
      u_id = query.arguments[0].value.value;
    }
  } else {
    u_id = u.id;
  }
  let id = directives.arguments[0].value.value;
  let relation = directives.arguments[1].value.value;
  let type = fieldDef.astNode.type.kind === 'ListType' ? fieldDef.type.ofType :
      fieldDef.type;

  return graph.db.all('SELECT ' + id + ' as id FROM ' + table + ' WHERE ' + relation
       + ' = ? ' + (limit ? ' LIMIT ' + limit : ''), u_id).then((result) => {
    return result.map(item => {
      return new Node(item.id, type);
    });
  });
};

var getRootNode = (graph, queryType) => {
  return new Node(0, queryType);
};

var nodeType = (graph, node) => {
  return node.table;
};

module.exports = {
  Node,
  getEdges,
  getRootNode,
  nodeType
};
```

Listing A.3: functions.js

B Appendix B

```
#/bin/bash
function prepare_file {
sed -i \
-e '1,$ s/character set utf8 collate utf8_bin//' \
-e '1,$ s/ ENGINE=InnoDB DEFAULT CHARSET=utf8//' \
-e '1,$ s/LOCK TABLES .* WRITE;//' \
-e '1,$ s/ALTER TABLE .* KEYS;//' \
-e '1,$ s/UNLOCK TABLES;//' \
-e '1,4 d' \
-e 's/INDEX USING BTREE (.*),\?//' \
-e '/^\s*$/d' \
$1
sed -i -n -e '1h;1!H;${g;s/\,\n);/\n);/g;p}' $1
}
prepare_file 01ProductFeature.sql
prepare_file 02ProductType.sql
prepare_file 03Producer.sql
prepare_file 04Product.sql
prepare_file 05ProductTypeProduct.sql
prepare_file 06ProductFeatureProduct.sql
prepare_file 07Vendor.sql
prepare_file 08Offer.sql
prepare_file 09Person.sql
prepare_file 10Review.sql
touch database.db
sqlite3 database.db < 01ProductFeature.sql
sqlite3 database.db < 02ProductType.sql
sqlite3 database.db < 03Producer.sql
sqlite3 database.db < 04Product.sql
sqlite3 database.db < 05ProductTypeProduct.sql
sqlite3 database.db < 06ProductFeatureProduct.sql
sqlite3 database.db < 07Vendor.sql
sqlite3 database.db < 08Offer.sql
sqlite3 database.db < 09Person.sql
sqlite3 database.db < 10Review.sql
```

Listing B.1: preparefiles.bash

```sql
DROP TABLE IF EXISTS `knowsperson`;
CREATE TABLE `knowsperson` (
  `person` int(11) not null,
  `friend` int(11) not null,
  PRIMARY KEY (person, friend)
);
INSERT INTO `knowsperson` VALUES (1,2),(1,3),(1,4),(1,5),(1,6),(2,7),(3,7),(4,7)
    ,(5,7),(6,7),(7,8),(7,9),(7,10),(7,11),(7,12),(8,13),(9,13),(10,13),(11,13)
    ,(12,13),(13,14),(13,15),(13,16),(13,17),(13,18),(14,19),(15,19),(16,19),(17,19)
    ,(18,19),(19,20),(19,21),(19,22),(19,23),(19,24),(20,25),(21,25),(22,25),(23,25)
    ,(24,25),(25,26),(25,27),(25,28),(25,29),(25,30),(26,31),(27,31),(28,31),(29,31)
    ,(30,31);
```

Listing B.2: knowsperson.SQL

```javascript
var fs = require('fs');
var {
  rawRequest
} = require('graphql-request');
const args = require('minimist')(process.argv.slice(2));

const {
  queries1,
  queries2,
  queries3,
  queries4,
  queries5
} = require('./queries');
var _ = require('lodash');

const sendRequest = (iter, query) => {
  return rawRequest('http://localhost:4000/graphql', query).then(({
    data,
    extensions
  }) => {
    totalCalc += extensions.tracing.calculation.duration;
    totalExec += extensions.tracing.execution.duration;
    if (--iter > 0)
      return sendRequest(iter, query);
  });
};

const execute = (iter, q) => {
  return sendRequest(i, q[iter]).then(() => {
    console.log('Query ' + iter + ' executed');
    logger.write(iter + '     | ' + totalCalc / i + ' | ' + totalExec / i + '\n');
    if (++iter < _.size(q)) {
      totalCalc = 0;
      totalExec = 0;
      return execute(iter, q);
    }
  });
};

const setupWriteStream = (file, i) => {
  var logger = fs.createWriteStream(file, {
    flags: 'a'
  });
  logger.on('error', function(err) {
    console.error(err);
  });
  logger.on('open', () => {
    console.log('Starting tests with ' + i + ' iterations.');
  });
  logger.on('finish', () => {
    console.log('Tests completed, output available in output.txt.');
```

```
  });
  return logger;
};

if (!(typeof args.i === 'number')) {
  console.error('Missing or wrong type for argument, please provide a number of
      iterations to be run!');
  console.error('Usage: "node logger.js -i iterations"');
} else {
  var i = args.i;
  var logger = setupWriteStream('output.txt', i);
  var totalCalc = 0;
  var totalExec = 0;
  logger.write(
    'TEST RESULT FOR FILE: queries.js\nAVERAGE TIMES IN NS\n
        ================================\nQuery | Calculation | Execution\n'
  );
  execute(0, queries1).then(() => {
    logger.write('================================\n');
    logger.end();
  });
}
```

Listing B.3: logger.js

```
//CYCLIC
var queries1 = [
  `{Product(nr:6){reviews{reviewFor{label}}}}`,
  `{Product(nr:6){reviews{reviewFor{reviews{reviewFor{label}}}}}}`,
  `{Product(nr:6){reviews{reviewFor{reviews{reviewFor{reviews{reviewFor{label
      }}}}}}}}`,
  `{Product(nr:6){reviews{reviewFor{reviews{reviewFor{reviews{reviewFor{reviews{
      reviewFor{label}}}}}}}}}}`,
  `{Product(nr:6){reviews{reviewFor{reviews{reviewFor{reviews{reviewFor{reviews{
      reviewFor{reviews{reviewFor{label}}}}}}}}}}}}`,
  `{Product(nr:6){reviews{reviewFor{reviews{reviewFor{reviews{reviewFor{reviews{
      reviewFor{reviews{reviewFor{reviews{reviewFor{label}}}}}}}}}}}}}}`
];

//ACYCLIC
var queries2 = [
  `{Person(nr:1){knows{knows{name}}}}`,
  `{Person(nr:1){knows{knows{knows{name}}}}}`,
  `{Person(nr:1){knows{knows{knows{knows{name}}}}}}`,
  `{Person(nr:1){knows{knows{knows{knows{knows{name}}}}}}}`,
  `{Person(nr:1){knows{knows{knows{knows{knows{knows{name}}}}}}}}`,
  `{Person(nr:1){knows{knows{knows{knows{knows{knows{knows{name}}}}}}}}}`,
  `{Person(nr:1){knows{knows{knows{knows{knows{knows{knows{knows{name}}}}}}}}}}`
];

//VARYING FIELDS
var queries3 = [
  `{Person(nr: 2) {reviews {reviewFor {offers {vendor {nr}}}}}}`,
  `{Person(nr: 2) {name reviews {title reviewFor {label offers {price vendor {
      country nr}}}}}}`,
  `{Person(nr: 2) {name country reviews {title reviewDate reviewFor {label comment
      offers {price deliveryDays vendor {country nr label}}}}}}`,
  `{Person(nr: 2) {name country nr reviews {title reviewDate nr reviewFor {label
      comment nr offers {price deliveryDays nr vendor {country nr label homepage
      }}}}}}`
];

//EXTREME BLOWUP
var queries4 = [
  `{Product(nr: 2) {productFeature {label}}}`,
  `{Product(nr: 2) {productFeature {products {label}}}}`,
```

```
  `{Product(nr: 2) {productFeature {products {productFeature {label}}}}}`,
  `{Product(nr: 2) {productFeature {products {productFeature {products {label}}}}}}`
];

//EXTREME BLOWUP, LIMIT 3
var queries5 = [
  `{Product(nr: 2) {productFeature(limit: 3){label}}}`,
  `{Product(nr: 2) {productFeature(limit: 3){products(limit: 3){label}}}}`,
  `{Product(nr: 2) {productFeature(limit: 3){products(limit: 3){productFeature(limit
      : 3){label}}}}}`,
  `{Product(nr: 2) {productFeature(limit: 3) {products(limit: 3) {productFeature(
      limit: 3) {products(limit: 3) {label}}}}}}`,
  `{Product(nr: 2) {productFeature(limit: 3) {products(limit: 3) {productFeature(
      limit: 3) {products(limit: 3) {productFeature(limit: 3){label}}}}}}}`,
  `{Product(nr: 2) {productFeature(limit: 3) {products(limit: 3) {productFeature(
      limit: 3) {products(limit: 3) {productFeature(limit: 3){products(limit: 3){
      label}}}}}}}}`
];

module.exports = {
  queries1,
  queries2,
  queries3,
  queries4,
  queries5
};
```

Listing B.4: queries.js