Haaga-Helia
University of Applied Sciences

How fast GraphQL is compared to REST APIs

Camille Oggier

Bachelor's Thesis
Degree Programme in BITE
2020

Haaga-Helia
University of Applied Sciences

Abstract

Date

Author(s)
Camille Oggier

Degree programme
BITE

Report/thesis title
How fast GraphQL is compared to REST APIs

Number of pages and appendix pages
36 + 4

These days GraphQL is getting bigger and more people are adopting this technology. Nevertheless, most of them do not know what the pros and cons of this new query approach in comparison to its ancestor RESTful API are.

The central concept of this research thesis is to put GraphQL on different test benches and compare relevant metrics to well-established technologies in order to answer a research question: Is GraphQL faster and better optimised than REST?

With the perspective of answering this question, we will use a quantitative approach that will lead us to: create a benchmark protocol to measure metrics, construct an analysis methodology to collect and compare results and finally offer a potential conclusion from our findings.

Keywords
GraphQL, REST, API, React, Web

Table of contents

Table of figures

Terms and abbreviations

REST:	Representational state transfer
SQL:	Structured Query Language
API:	Application Programming Interface
SOAP:	Simple Object Access Protocol
HTTP:	Hypertext Transfer Protocol
SMTP:	Simple Mail Transfer Protocol
FTP:	File Transfer Protocol
ACID:	Atomicity, Consistency, Isolation, Durability
PHP:	Personal Home Page
TCP:	Transmission Control Protocol
NoSQL:	No Structured Query Language
JSON:	JavaScript Object Notation
DOM:	Document Object Model
UI:	User Interface
CDN:	Content Delivery Network
Mbps:	Megabit per seconds

1 Introduction

The purpose of this research thesis is to help any entity interested in GraphQL to get a first overview of the technology and answer a commonly asked question: Is GraphQL faster and better optimised than REST? In order to achieve that, this thesis will give access to a tool that will help to measure the potential gain of using GraphQL over its old predecessor: the traditional REST standard.

First of all, it is required to dive into query standards' history to understand the needs of such a new mechanism. Afterwards, it is essential to describe how GraphQL operates to have a better understanding of the technology. Then, finally, we will explain and run a few benchmarks on a dedicated developed program in order to get measurements and formulate our conclusions.

1.1 Context of research

With the aim of answering a specific question, this memoir will first expose the motivations and the needs for this research thesis. This will attempt to give you a comprehensive understanding of the background for this project.

1.1.1 Personal background

Self-taught developer and entrepreneur from an early age, I had the opportunity to start my career in information technology with IBM Switzerland. After four years of working for this renown organisation, I acquired a lot of experience on how big companies have to deal with technology choices. From my departure from IBM to now, I worked as a full-stack freelancer. This position gave me a precise picture of the current market, and I could attest the rise of the GraphQL demand over the past three years.

1.1.2 Motivations

The need for researching and obtaining numbers about this technology comes from a professional background. As a freelance full-stack developer, we always advise clients who have no idea about the solution they need, and this is where freelancers face a significant problem. Natively, it is obvious to recommend technologies that we are already familiar with, but this proposal does not always rely on factual arguments that will benefit client's product. Even if this suggestion is perfectly matching the needs, it is always a painful process to justify choices to people that do not necessarily have the knowledge to understand those arguments.

The case mentioned above is a widespread story, especially when it concerns new technologies like GraphQL. Therefore, this is where comes the approach of measuring the performance of each technology to help the decision process. With those metrics, engineers become able to deliver to the client concrete projections of their future application's performances and costs which are solid arguments that they may quickly assess.

1.1.3 State of the art

As of May 2020, you can find many comparisons of REST and GraphQL on the internet. They mostly expose the pros and cons of using both technologies on a theoretical level. But it is almost impossible to find real proofs of these arguments except if you try them by yourself. Moreover, there is no methodology to compare those two standards that are clearly explained and reproducible.

1.1.4 Research gap

What cannot be found about this technology is a quantitative approach for their comparison. Translate those theoretical approaches of the two standards to concrete and numbered observations for their advantages and disadvantages.

1.2 Objectives and outcomes

The main objective of this research thesis is to answer this central question:

Is GraphQL faster and better optimised than REST?

In order to answer this question, the quantitative research approach will be used. The quantitative research uses numbers and measurements to connect the empirical observations to a potential unbiased answer (Wikipedia, 2020). To collect those numbers, a reproducible methodology will be provided so that anyone can use it. The methodology will be in the form of a benchmark tool that will be open-source and available for free.

1.3 Research delimitations

The projections and conclusions of these researches are limited to a small number of general use cases. An infinity of possibilities and contexts can be imagined to perform measurements, but it would not be relevant to represent all of these alternatives in this thesis.

As a result, this thesis will stick to some traditional and modern examples that symbolise the average case with GraphQL. Any entity that cannot find their use case represented

can have access to the open-source tool and perform their benchmark and projections using the fully explained methodology in the next chapters.

2 The technology

The next chapters will expose the theoretical part of this thesis. It will explain the history of both technologies and their mechanism.

2.1 History of query standards

First of all, it is necessary to define what query standards are concerned. In computer science, there is a wide variety of query standards that apply to many environments, such as, Structured Query Language (SQL), Graph Query Language, Representational State Transfer (REST) and so much more. They all have in common the goal to bring the ability to a system to query another and access distant data over a standard. This standard can rely on a furnished communication panel as they can use different protocols to operate. As you can tell, query standard is a quite broad concept, so our theoretical part will reduce the scope of definition to the one we are interested in: the web API standards.

2.1.1 REST's history

Back in the 2000s, there was no real standard on how to create and use API. Protocols like SOAP were required, but back in the days, they remained very complex and challenging to use. The priority was to create flexible architectures that can rely on many communications protocols like HTTP, SMTP, FTP and so on. As a consequence, the need for a lightweight alternative rose with the appearance of the web demand.

The year 2000 will change the API's landscape forever. A researcher from the University of California, Roy Thomas Fielding, defines for the first time the concept of REST API in his doctoral thesis published later this year. The outcomes of this new standard are clear, Fielding says:

« REST is a coordinated set of architectural constraints that attempts to minimise latency and network communication while at the same time maximising the independence and scalability of component implementations. »
(Fielding, 2000, p. 148)

The new standard will try to get rid of all the superfluous metadata that englobe the SOAP protocol. SOAP and REST are commonly compared to the envelop and the postcard.

REST is a much lighter and optimised architecture compared to SOAP. That being said, the need for a thinner standard comes at a cost. In fact, REST will have to reduce the number of options and capabilities that he has compared to SOAP. As a consequence, REST does not intend to replace SOAP permanently that still fulfil many use cases. As an example, this table reflects the major differences between those two standards.

SOAP vs. REST Comparison: Which is Right for You?

Difference	SOAP	REST
Style	Protocol	Architectural style
Function	Function-driven: transfer structured information	Data-driven: access a resoruce for data
Data format	Only uses XML	Permits many data formats, including plain text, HTML, XML, and JSON
Security	Supports WS-Security and SSL	Supports SSL and HTTPS
Bandwidth	Requires more resources and bandwidth	Requires fewer resources and is lightweight
Data cache	Can not be cached	Can be cached
Payload handling	Has a strict communication contract and needs knowledge of everything before any interaction	Needs no knowledge of the API
ACID compliance	Has built-in ACID compliance to reduce anomalies	Lacks ACID compliance

Visit upwork.com to learn more
©2018 Upwork Inc.

upwork

Figure 1 SOAP vs. REST: The key differences (Wodehouse, 2017)

A significant loss of REST over SOAP is the lack of ACID compliance. ACID is a common concept in the database world that aims to guarantee the Atomicity, Consistency, Isolation and Durability of data. In some cases, sensitive information requires such compliance, so if you decide to use REST, you will have to implement that compliance layer by yourself.

To conclude, we can say that REST came to solve problems that SOAP could not handle. At a time where bandwidths were very limited and coupled with the explosion of the web, REST appeared like a lifesaver and made our web future a reality.

2.1.2 GraphQL's history

Compared to REST, GraphQL is much younger than its old predecessor. The project was internally initiated by Facebook in 2012 to solve a problem related to the rising demand for mobile applications. At the beginning of the smartphone ecosystem, devices were very limited by their connectivity. Applications required to have the fewest request possible to optimise speed and load time. But companies like Facebook with rich applications and news feed were struggling at reaching those criteria as they needed multiple queries in order to fetch all pieces of information related to a post, for example. As a consequence, Facebook decided to create a new query standard that will allow them to wrap all data needed into a single query.

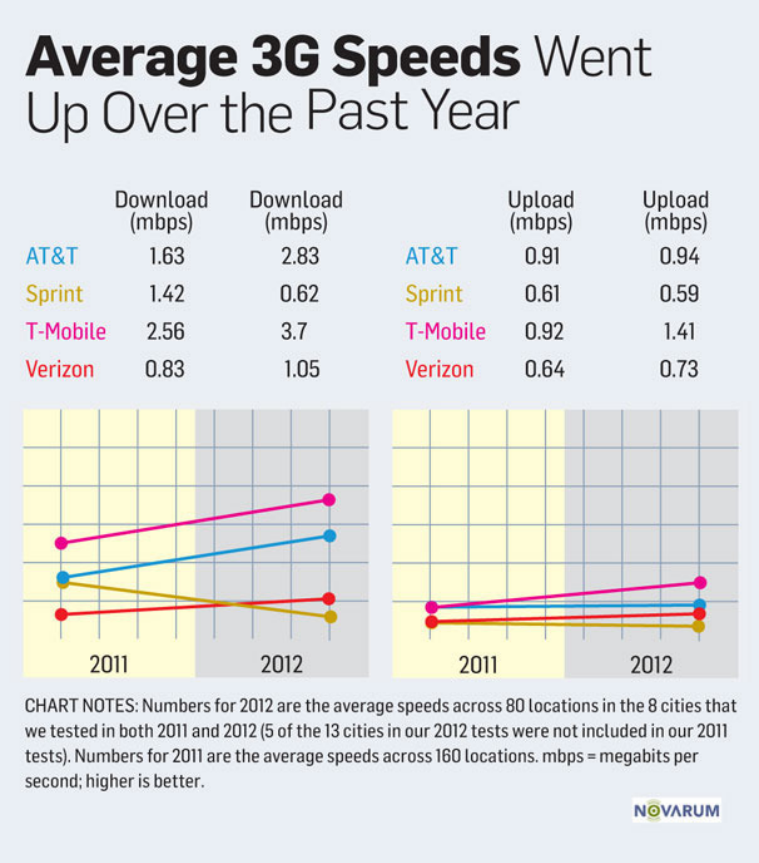


Figure 2 Average 3G Speeds (Novarum & PCWorld, 2012)

This figure shows the average internet speed reached by users from various internet providers across eighty locations in the United States in 2011 and 2012. As you can tell, at that time, connectivity was minimal, and every query was a tremendous effort for the network.

The project remained private until 2015, the year Facebook decided to publicly release this new technology. At the very beginning, people at Facebook were very sceptical about releasing GraphQL to the community, as it was just a bunch of PHP code. But they realised that GraphQL is so much more than just a tool that helps their company to retrieve complicated things for their front-end. In addition, two years earlier, Facebook released its first significant contribution to the open-source community, which was ReactJS.

To bring popularity to GraphQL, Facebook started to build a NodeJS based version of GraphQL and proposed to the community to build variants in different languages, in order to make this technology accessible for every established infrastructure.

This scepticism phase behind, Facebook took one more significant step into the open-source community by detaching the initial project from the mother company in 2018 and by gathering every sub-community into a single foundation, the GraphQL Foundation (Byron, 2019).

2.2 GraphQL Architecture

GraphQL is very intuitive to use. It works on schema-based request, which makes it easy to read and build. This technology also avoids tons of logic in order to restructure the data after multiple fetches. We will discuss those concepts later into the implementation part. But first, we will have a look at the architecture and compare it to a REST API construction.

2.2.1 Simple architecture

The most common architecture is a simple client-server architecture. This architecture features a single server instance of GraphQL specifications. This server is connected to the database and will simply answer the request made to the endpoint.

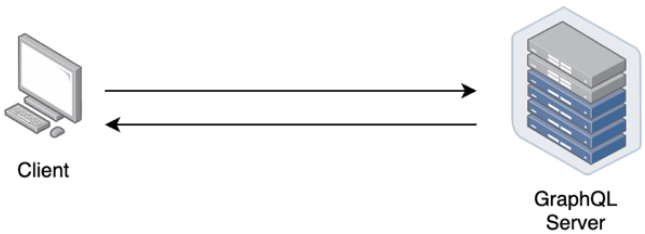


Figure 3 A simple client-server architecture with GraphQL (How to GraphQL, 2020)

The connection between the server and the client can be made on any protocols. That means the transport may be done over WebSockets, TCP and many more options. It is

also relevant to notice that the GraphQL server can work with any type of database, whatever it is SQL or NoSQL. The developed resolvers are able to serve any request in a wide variety of environments.

2.2.2 Gateway architecture

The gateway architecture is the best representation of the first use-case made by Facebook back in 2012. The idea is that the GraphQL server will interact with multiple services like REST endpoints, external services, other GraphQL instance, etc. To deliver one single endpoint to query.

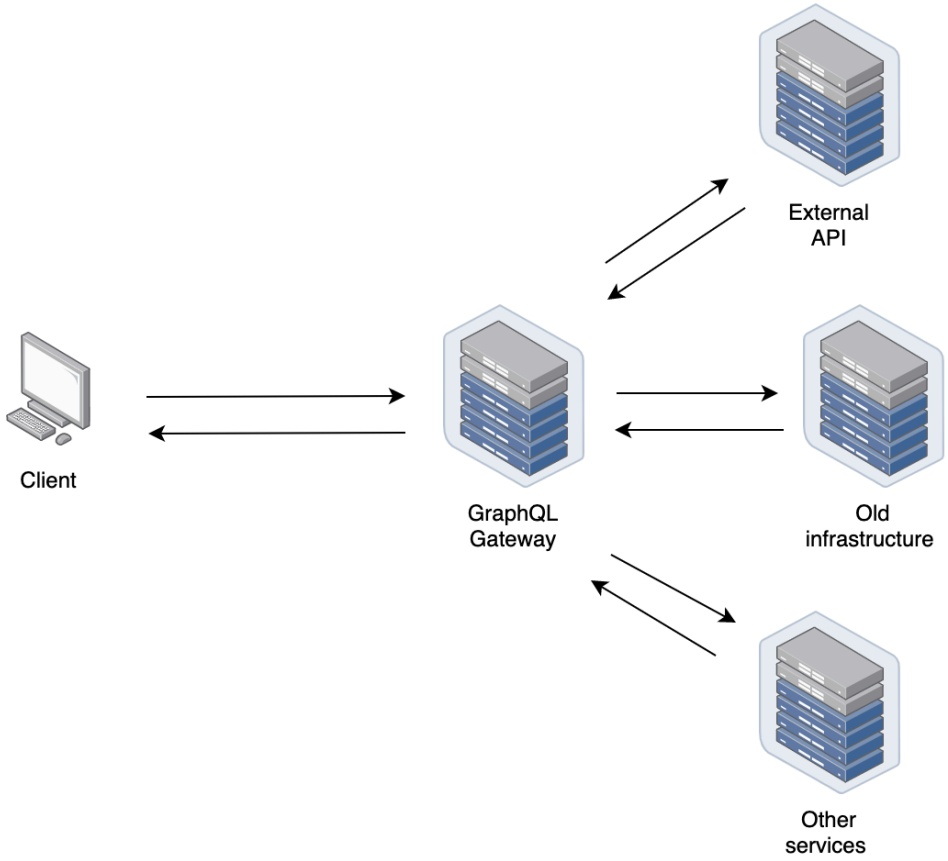


Figure 4 GraphQL used as a gateway for complex sub-systems (How to GraphQL, 2020)

This configuration is very convenient and reduces the amount of complexity for querying the backend from the front-end. It is also brilliant because it allows us to integrate this new technology while keeping the old infrastructure in place. This concept makes GraphQL very flexible as adopter do not have to refactor the whole system.

In this case, the GraphQL gateway is responsible for making every sub-system collaborate together and to resolve which necessary service is needed to be called to serve the request.

2.2.3 Hybrid architecture

To conclude this chapter, the hybrid architecture reflects the combination of both gateway and simple configuration.

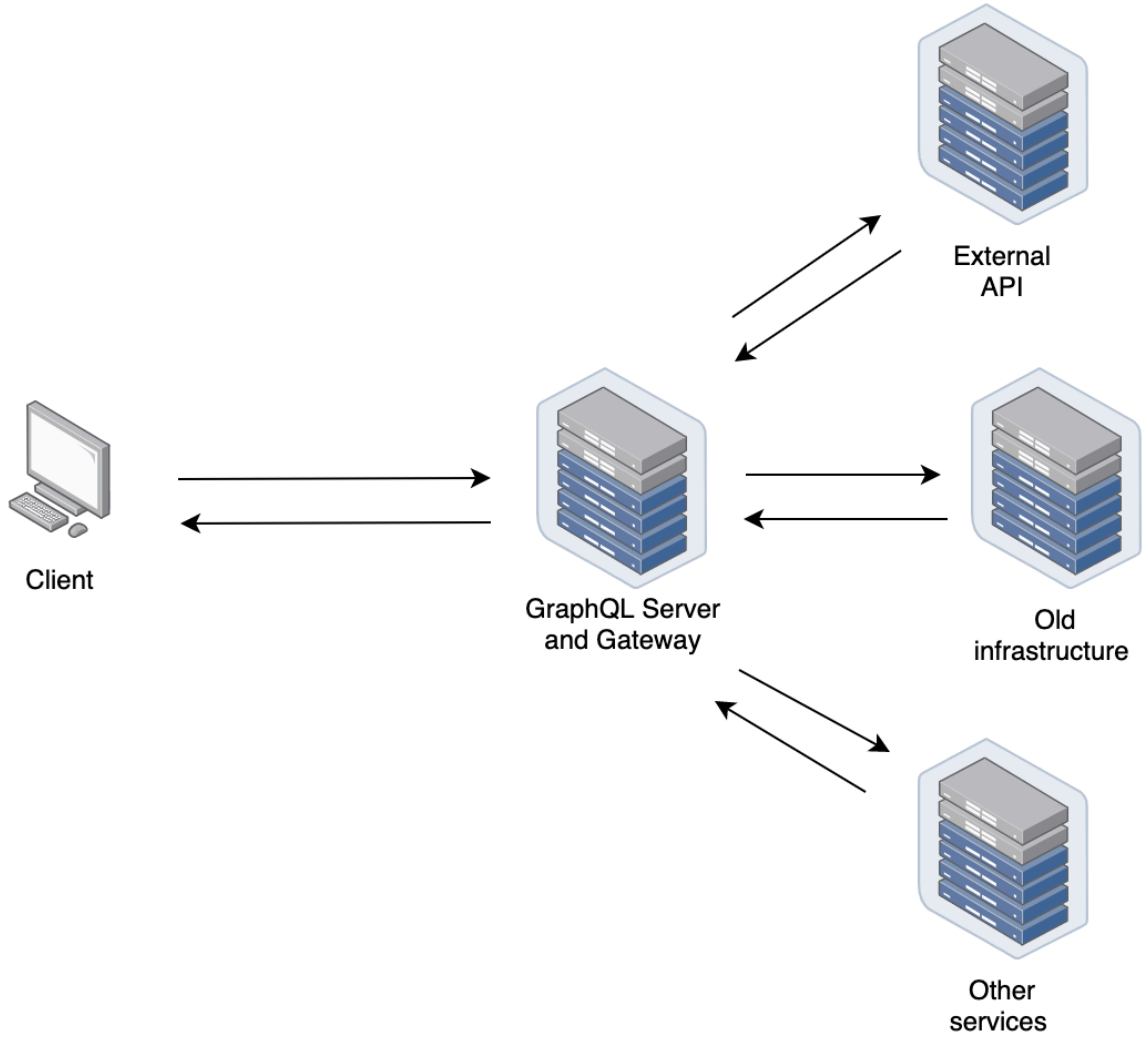


Figure 5 Hybrid GraphQL architecture (How to GraphQL, 2020)

In this architecture, the gateway is placed in the centre of the system as the primary end-point. This gateway has a database directly connected to it as a simple architecture. One of the main reasons for adopting such structure is the case where any of the architectures exposed in the chapters above has been used previously for implementing GraphQL to the system. Then the need for extending features will very often result in the use of a hybrid configuration.

2.3 Difference between GraphQL and REST architecture

The core difference between those two methods is the number of endpoints available. While GraphQL tries to gather every request to a single spot, REST is made so that every resource is behind a specific endpoint.

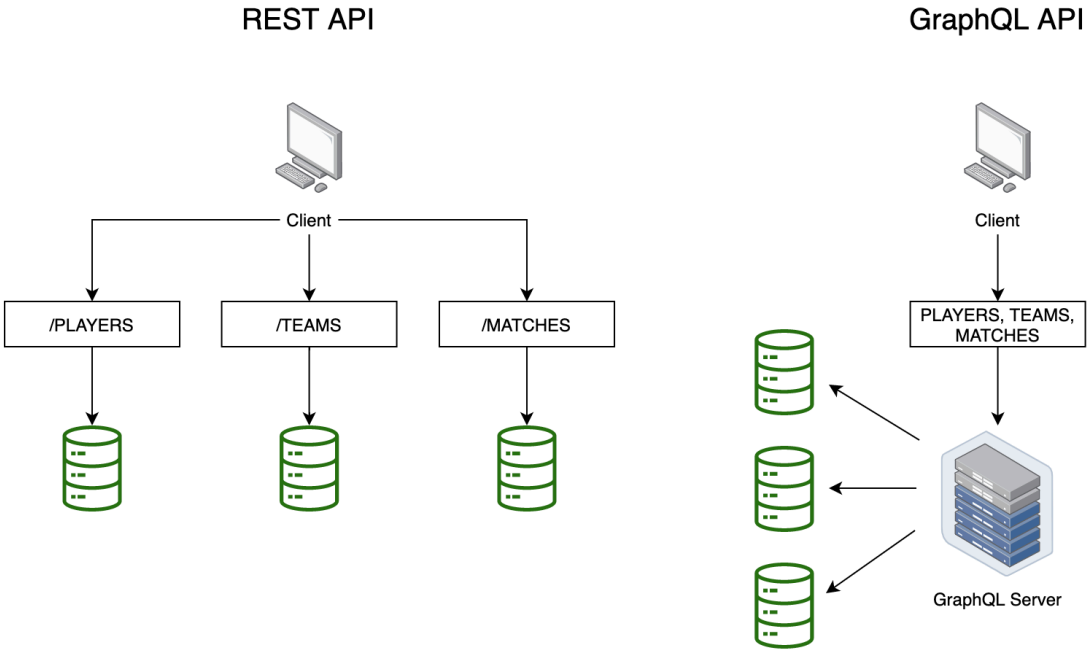


Figure 6 A simple diagram illustrating structure differences (Jung, 2019)

In the case of the REST configuration, it is up to the front-end to adapt and query the right endpoint for the desired resource. This can lead to tricky situations where any changes to the backend can result in modifying the front-end as well.

2.4 GraphQL implementation

As mentioned earlier, GraphQL is effortlessly accessible, at least for a basic implementation. So, in the next chapters, we will have a look at the basics of GraphQL to give the reader an essential understanding of how to implement it concretely.

In the following order, the next chapters will discuss how the front-end proceed to request data to the GraphQL server. Finally, the last section will observe how the server resolves and serves the request.

2.4.1 Front-end clients

First of all, the front-end needs a client to interact with the backend server. There are plenty of different solutions on the market that may work better with specific technologies. Here is a brief list of significant clients:

- **Apollo Client**: probably the most famous JavaScript client.
- **Apollo Fetch**: a lightweight version of the main Apollo client.
- **FetchQL**: a minimal client that can be used in any JavaScript case.
- **GraphQL Request**: which is a little more convenient, but still lightweight.
- **Relay Modern**: an improved iteration of the first Relay client built by Facebook at the very beginning of GraphQL.

The choice of the client may be crucial for an application, as each implementation may include or not some features. For example, GraphQL does not have native caching like REST architecture, because it does not rely on HTTP, which can offer the possibility of caching request automatically. Some clients, like Apollo, have an excellent implementation of caching, so it is essential to choose wisely the front-end client.

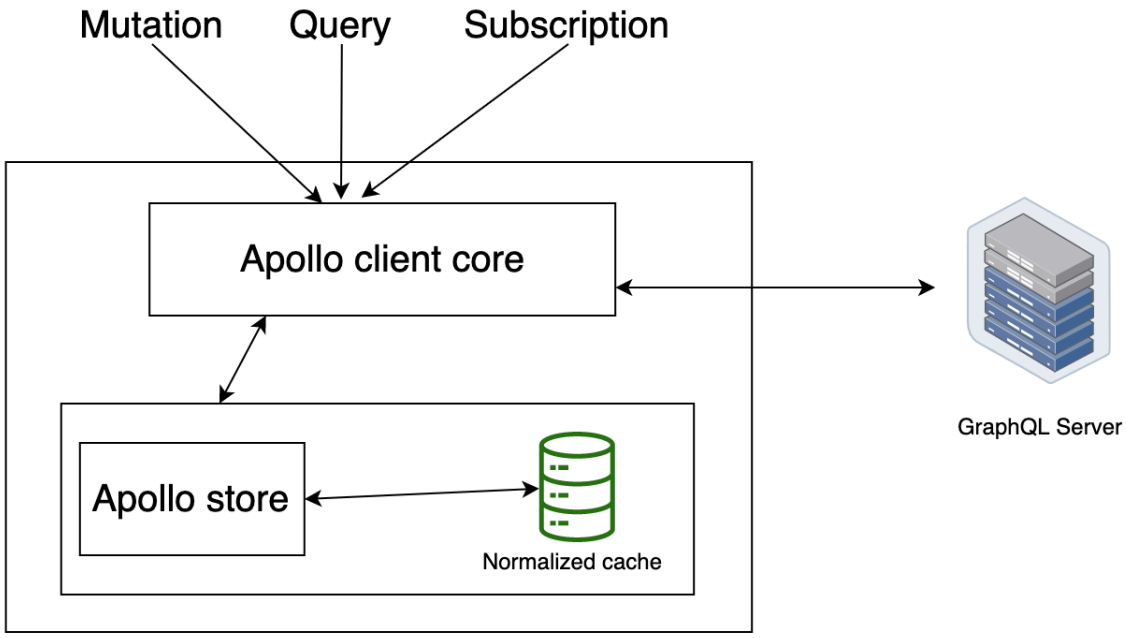


Figure 7 Apollo Client caching system (Cerroni, 2019)

The figure above illustrates where Apollo client stores the cached data from its data store. In that way, the client will automatically observe any mutation to the data received from the backend compared to the data already present in the store. In case of a mutation, the cached data will be updated.

2.4.2 Front-end requests

The following example demonstrate how a developer can formulate a request to the GraphQL server and see what the answer from the backend looks like.

As previously mentioned, GraphQL is a schema-based standard. This means data will be mapped as a set of schemas in the backend. In order to query and specify what information is needed from the front-end, the programmer develops a query the same way. A basic query will comport all fields needed from an entity following the same hierarchy as the query type specified on the GraphQL server. (GraphQL Foundation, 2020)

```
{
  hero {
    name
    appearsIn
  }
}
```

Figure 8 A basic example of a query syntax (GraphQL Foundation, 2020)

This figure is a simple example of a query that you can request from any GraphQL client. More precisely, in this case, schema reaches the hero query type. In this interface, there should be first a name field, and then an appearsIn field. With this syntax, the front-end is able to obtain only the required fields. As an example, considering that a hero may have more fields that the client is not interested in, a REST architecture would result in fetching and storing unnecessary data which is commonly called “over-fetching”. (Prisma, 2017)

The result will be presented as a JSON format and will follow the query structure.

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ]
    }
  }
}
```

Figure 9 A basic query result (GraphQL Foundation, 2020)

Experimentations can be expanded on more complex requests. In fact, this query standard is mighty and accepts a vast number of options. The next example features a search demonstration.

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
    ... on Starship {
      name
      length
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo",
        "height": 1.8
      },
      {
        "__typename": "Human",
        "name": "Leia Organa",
        "height": 1.5
      },
      {
        "__typename": "Starship",
        "name": "TIE Advanced x1",
        "length": 9.2
      }
    ]
  }
}
```

Figure 10 A search query on the left and its result (GraphQL Foundation, 2020)

In the case above, the client asks the GraphQL server to lookup on three types: Human, Droid and Starship. In each type, the resolving procedure will look for the string “an” and return the entry under the corresponding type name. In that way, you can build robust queries directly from the front-end.

2.4.3 Backend implementation

As for the front-end, the backend has many solutions for deploying and developing the GraphQL server. You can find plenty of implementations in many languages on the open-source market. For this reason, the next passage will discuss the general concepts shifted to the backend and show some basic examples in Node. Keep in mind that the following GraphQL is exhaustive and may change through time (GraphQL Foundation, 2020):

- JavaScript
  - **GraphQL.js:** this is a reference implementation for the backend.
  - **Express-graphql:** make the GraphQL server working over Express web server.
  - **Apollo-server:** one of the main references.
- Java
  - **Graphql-java:** the only java implementation mentioned on the official documentation.
- C# / .NET
  - **Graphql-dotnet**

The primary thing to do is to get on the schemas and types definition. Every entity has to be defined before any use; this is how you declare a contract of what is accessible between the backend and the front-end.

There are two primary types in every server implementation, which are:

- **Query:** this type will gather all the data fetching requests.
- **Mutation:** in contrast to the first type, it will contain all the modification requests (create, update and delete).

These two types will interact with secondary types that contain definitions of the stored data. As an example, implementation of types will be observed on an Apollo-server demo. This example features the two primary types coupled with other necessary types for a given application.

```
type Query {
  upcomingEvents: [Event]
}

type Event {
  name: String
  date: String
  location: Location
}

type Location {
  name: String
  weather: WeatherInfo
}

type WeatherInfo {
  temperature: Float
  description: String
}
```

Figure 11 Query type (Apollo GraphQL, 2020)

```
type Mutation {
  # This mutation takes id and email parameters
  updateUserEmail(id: ID!, email: String!): User
}

type User {
  id: ID!
  name: String!
  email: String!
}
```

Figure 12 Mutation type (Apollo GraphQL, 2020)

As you may have noticed, schemas are strictly typed. That means all fields will have to specify a primary scalar type or a custom type in order to pass through the validation process. Default types are the following (GraphQL Foundation, 2020):

- **Int:** A 32-bits signed integer.
- **Float**: signed double-precision floating-point value.
- **String:** UTF-8 character sequence.
- **Boolean:** true or false.
- **ID:** unique id that is serialised.

Once types are defined, it is mandatory to indicate how and where to get the data in order to feed the fields. this step is called resolving, so programmer will develop resolver functions. A resolver function can basically access everything everywhere. This makes your server able to interact with any data source, as previously discussed in the gateway and hybrid architecture chapters.

```
const resolvers = {
  Query: {
    user(parent, args, context, info) {
      return users.find(user => user.id === args.id);
    }
  }
}
```

Figure 13 Example of a simple resolver (Apollo GraphQL, 2020)

In this example of a resolver function, the function describes how a user will be returned from an array of users. The selection is based on its id where the find function will go through the users' array and find the matching content.

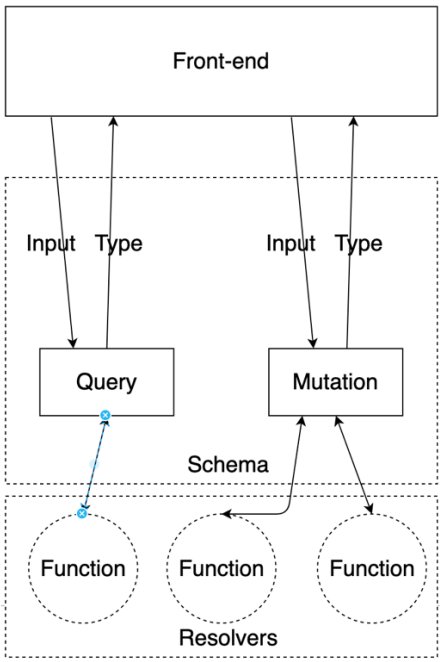


Figure 14 GraphQL server big picture (Rhyne, 2017)

To sum up, as shown on the big picture of the GraphQL server above, this technology can add more complexity to the backend side because of his ability to validate everything natively and his flexible architecture.

3 Benchmark tool

Theoretical comparison is essential to understand core differences between those two technologies, but our approach includes an actual test of the two standards side by side. In order to achieve that, a benchmarking tool is needed to compare our REST queries to the GraphQL ones.

As of now, no solution on the market can fulfil our needs. In fact, all solutions bring the ability to query and test an API, and most of them bring some performances indicators. But it is impossible to compare those insights side by side right after the fetch. To have a similar result, we could take those outputs and put them into a model that compares them. This alternative is not very suitable and easy to use, so for this significant reason, a benchmark program will be created and will contribute to the community.

In the next chapter, all the technologies and libraries used to create our software will be described. Lastly, we will examine the way the tool is designed and its mechanisms to process the requests.

3.1 Development tools and libraries

This development project aims first to be available publicly. With the goal of being accessible and used by anyone, the chosen libraries and tools will be powered by open-sourced contributions. The second fundamental criterion is the popularity of our choices. In effect, the more our libraries and technologies are used, the more maintenance and contributions the tool will receive from the open-source community.

3.1.1 React

React, also known as ReactJS, is a JavaScript library built and publicly released by Facebook in 2013. This library can be used to build a web application, and as his library designation suggests, it can be used aside other JavaScript libraries (Wikipedia, 2020).

React is an open-source project that initially had the purpose of creating a new approach for developing news feed on both Facebook’s platform: Facebook web and Instagram. The library is now one of the most used technology for creating user interfaces on the web but not only (Wikipedia, 2020). In 2015, Facebook brought React to mobile devices with React Native. This expansion of the React world encouraged many developers to adopt React for both web and mobile (Wikipedia, 2020).

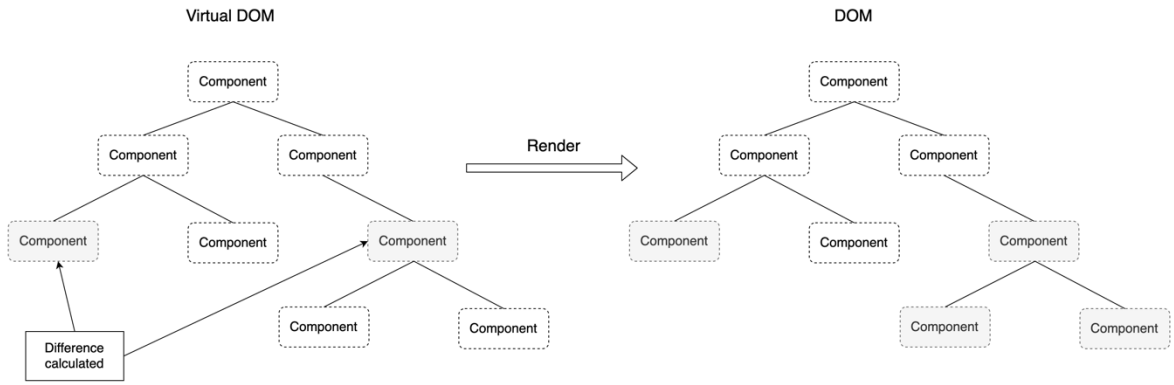


Figure 15 Main concept of React (Shakya, 2018)

React has a lot of pros, but one of the major advantages is the library's speed. React has been made to fasten the render process. With pure JavaScript, any modifications to the web page are directly rendered to the browser DOM (Document Object Model). This rendering process is burdensome for the browser as he has to reinterpret the whole document every time. To solve this issue, React has introduced his Virtual DOM. The process of rendering is significantly different as every modification is made to this virtual DOM before being evaluated. If a difference is detected between those two DOM, the change will be reflected in the real DOM. In that way, React reduces the number of renders in the real DOM and save plenty of resources (Shakya, 2018).

For all of these reasons, React is a perfect match for our project and will bring us the base structure of our benchmarking tool.

3.1.2 GraphQL Request

GraphQL Request or graphql-request, is one of the simplest and lightest GraphQL clients on the market. This client is very portable and can be implemented in every context. Even if this client has countless features missing for a real web application, its implementation fits our needs perfectly as we aim to perform simple GraphQL queries. It does also make a lot of sense to use it as it will be used next to the traditional native fetch for the REST request. In that way, we ensure that both protocols use the same number of features and do not have any advantages over the other.

3.1.3 Material-UI

Material-UI is a famous presentation framework for React. Material-UI has been created to give easy access to beautiful web components following the Material Design guideline made by Google. With over one million downloads per week, this highly accessible and open-source framework will serve our interface.

3.2 Implementation

The implementation phase is a crucial part for the reader in order to understand the methodology of the measurements. It is essential to explain the design process of the solution and discuss some choices.

3.2.1 Select metrics

Before diving into any type of implementation, we need to identify what are the key numbers to measure. These measurements will attest the precision of our results.

From a front-end side, JavaScript gets access to two different vital numbers for a request. The first number is the latency that can be measured by placing two timers before and after all the queries. With this technique, we will point how much time has passed between the start and the end of all requests. The second metric is the size of a response. By adding the byte size of every response, we can identify which API standard is the lightest in term of data transfer. However, the size measure will not reflect the total request size. Because of browser limitations, the front-end code does not have access to the size value of a query and its header, that means the tool is only able to calculate the size of the body. To sum up, you will have the following measurements:

- **Execution time**: identifies the fastest by measuring the time a standard will take to execute all requests.
- **Body size**: reveals the lightest by adding all body sizes for a given standard.

3.2.2 Measurement process

Once metrics were identified, we can start creating the algorithm diagram that will be implemented. With this representation of our code, we will know where to put measurements and make sure outputs are valid.

Before that, the capabilities of the tool will be need to be identified as this will influence all the related diagrams. To help to identify those, our needs will be expressed in the format of user's stories.

- As a user, I want to configure a sequence of API calls for both REST and GraphQL.
- As a user, I need the possibility to increase the number of iterations for each call in order to represent multiple users.
- As a user, I want the option to execute queries in parallel or in sequence.
- As a user, I need to observe the latency and the total size for each standard.

With a more unobstructed view of the goals, diagrams can be created.

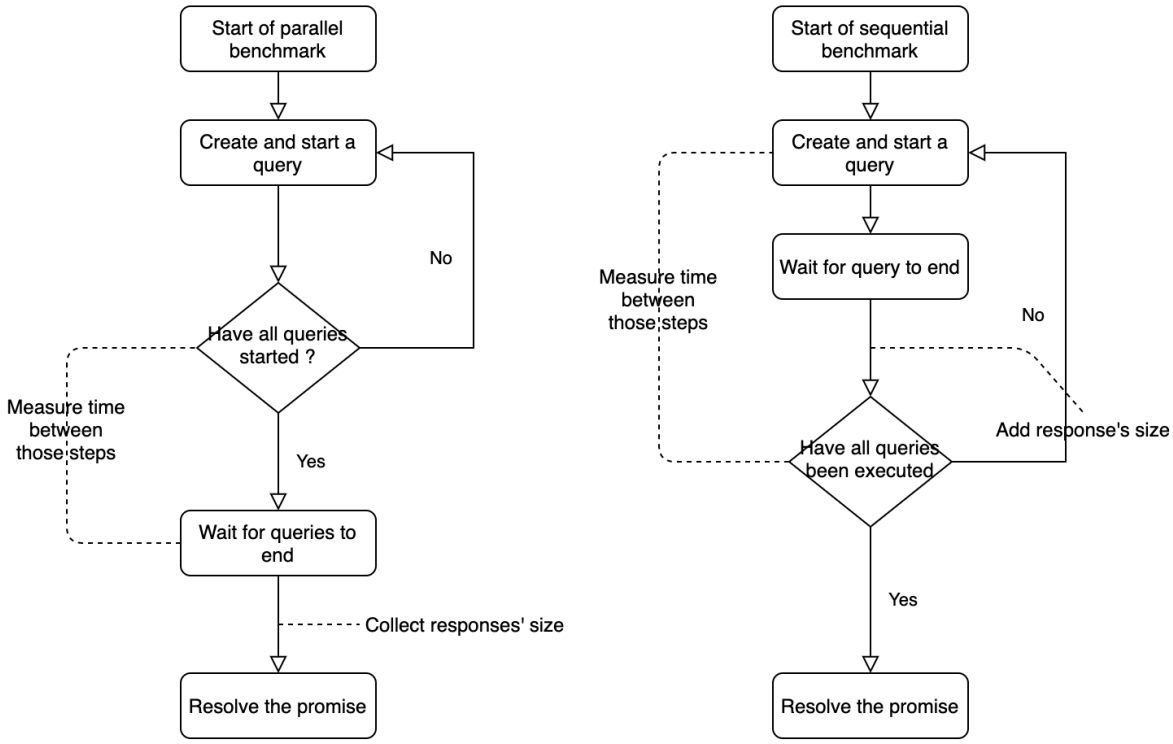


Figure 16 Diagram of the two benchmark sequences

With the diagram above, our different cases are covered. You can distinguish the two measurements indicated by dashed lines. We now precisely know how to proceed for our implementation on the algorithmic side.

3.2.3 User interface

The design question may appear unnecessary, but a great design will make the use and the presentation of results clearer. This will also appeal to the community's interest and may attract contributors.

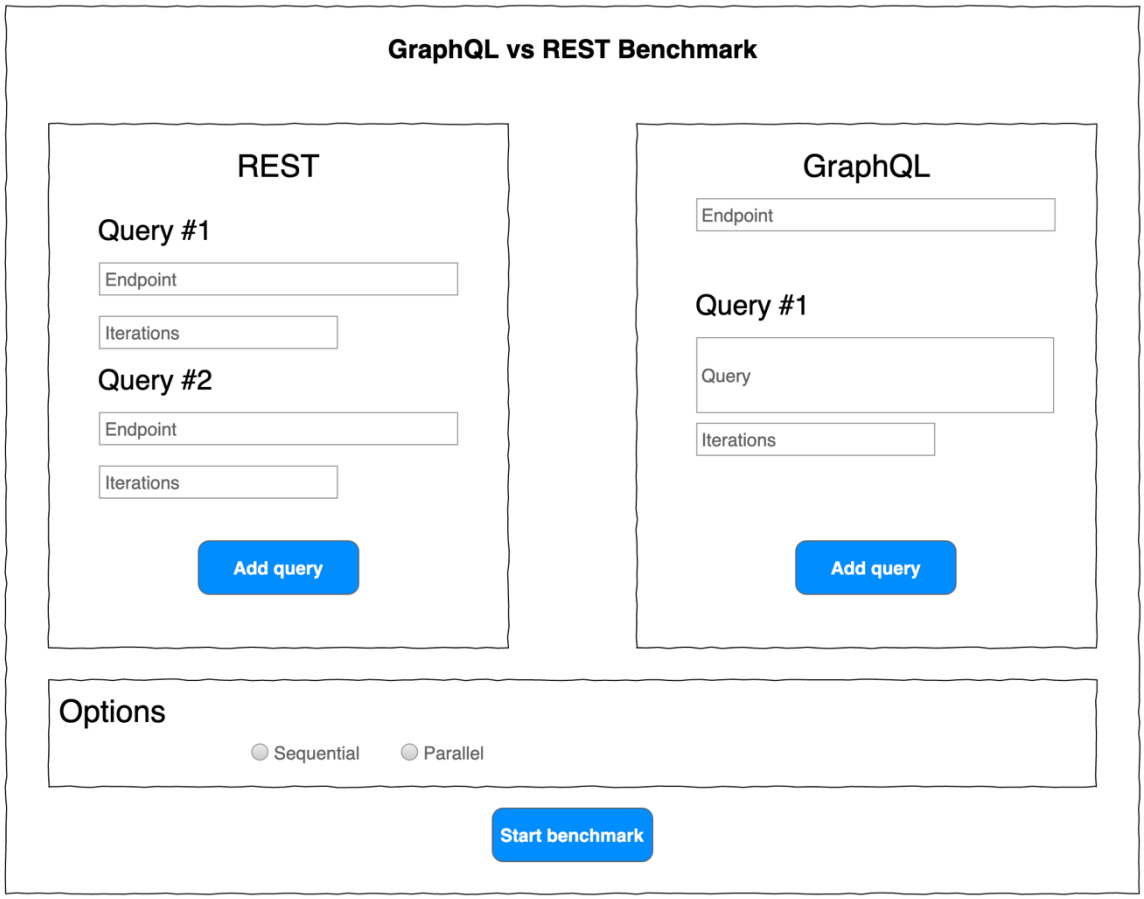


Figure 17 Mock-up for the web interface

It is crucial to keep the interface as simple as possible. This application does not intend to provide a rich UI, so the interface will keep it clear with a side by side layout that makes a lot of sense in a comparative context.

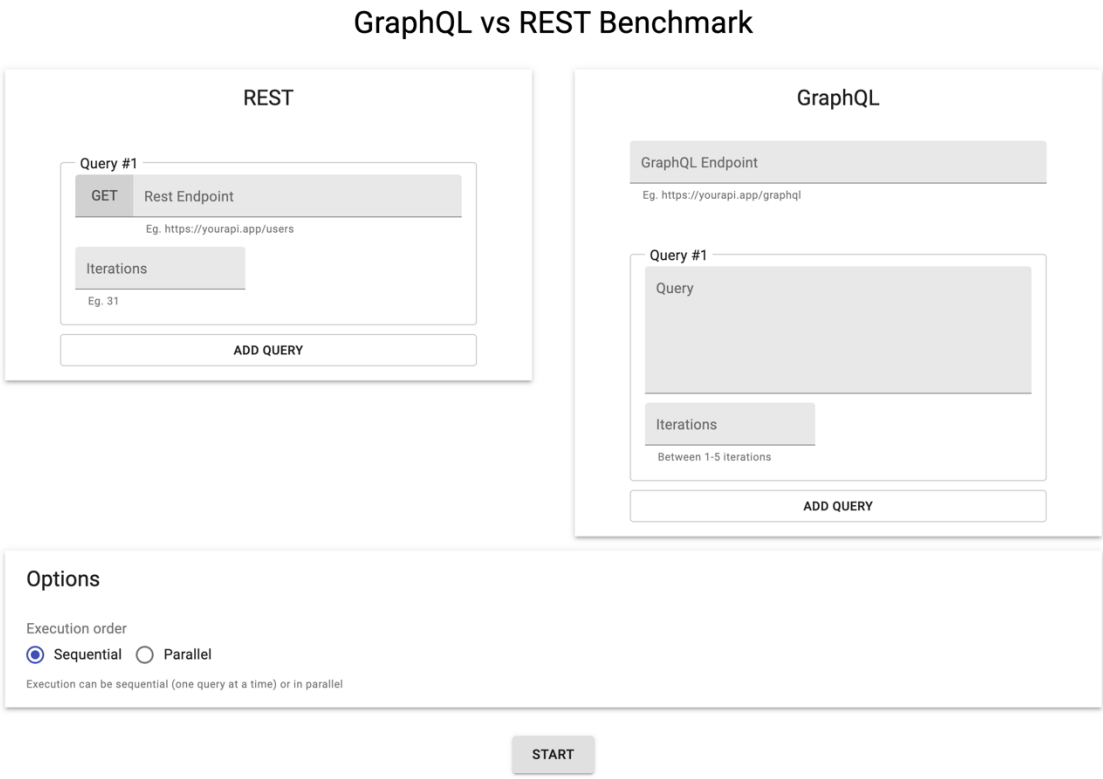


Figure 18 React implementation of the mock-up

As you can observe, the real implementation is really close to the mock-up. It does use Material-UI as mentioned earlier to bring a clean look to the design. We will expand the design topic in the next chapter that will explain the display of results.

3.2.4 Results display

The whole process of creating a benchmark interface would not make sense if results are poorly presented. The target is to present our two standards side by side for every metric. Every measurement will feature a graphic representation to help the user to identify clearly the proportion of each result at first sight. The user will also access the precise value measured to exploit those outputs for further uses.

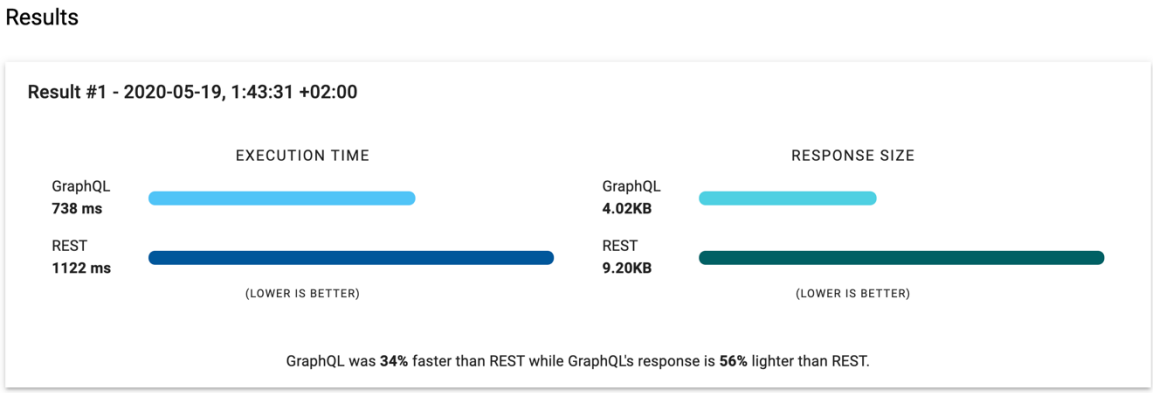


Figure 19 Example of results for the tool

As you can see in the figure above, every result logs the end time of each benchmark execution. This information may be required as a piece of context information to indicate to the user that the test was performed at this exact moment while the network and servers were experiencing a particular situation at the given time. This is something to take into consideration while analysing the results. Moreover, we can observe a small sentence at the very bottom that explains those results into meaningful words for non-technical users.

It is crucial to attract your attention to the chosen units for the presented numbers. To represent the time, the tool will stick to the original measured unit, which is milliseconds. This unit ensures that our measure keeps its precision as milliseconds are commonly used in the digital world. In addition, a gap of only 100 ms is already a considerable change to the result. Moving on the response size, the base unit is the Byte. As this value may vary from a very low to a high amount of data, the presentation layer will adapt and show any higher value to 1024 Bytes to KB and so on.

3.3 Tool's availability

This program aims to be available to any entity who needs to perform tests and benchmarks for those two technologies. To make it possible, the source code is published, and a published continuously build can be accessed online.

3.3.1 GitHub

GitHub is the biggest collaborative platform that hosts over 28 million of public repositories, with significant open-source projects like React or GraphQL. GitHub provides a set of tools to manage your source code with Git and collaborate with other people from your team or from the community (Wikipedia, 2020).

The benchmarking tool is part of this big platform and offers the opportunity to anyone to bring a contribution and make this program evolve. You can access the repository by following this link: https://github.com/camogg/graphql-benchmark.

3.3.2 Ready to use build

To bring more accessibility to this tool, a built version is hosted on a Google CDN. So people that may just want to give a try can access this application in one click. The online version can be found at the following address: https://graphql.oggier.dev/.

4 Analysis

With the help of our technical methodology described in the third chapter above, we are now able to dive into our analysis that will bring pieces of evidence to answer the central question of our research thesis.

Firstly, analysis methodology will be explained before exploring two situations from which data will be collected and discussed based on their results.

4.1 Analysis methodology

It comes as no surprise that the analysis methodology is an essential topic that needs to be described explicitly and clearly. In that way, we will ensure that our conclusions are made upon specific and defined conditions that attest the non-biased aspect of our opinion. It also allows the reader to reproduce and prove the veracity of our findings.

Our method comes with four critical steps:

1. Defining technical characteristics of the test.
2. Collecting samples of result from the benchmark procedure.
3. Merging samples to increase the accuracy of the results.
4. Comparing and discussing the results.

In order to make those steps clearer for the reader, the next sub-chapters will describe them more precisely.

4.1.1 Technical characteristics

The technical characteristics of a test are regrouping many specifications. From the environment's aspects to the configuration of the tool, each tested situation needs to expose its conditions to make the results reproducible.

Table 1 Technical characteristics template

Characteristic	Value
GraphQL backend technology	Technology used to deploy the GraphQL endpoint.
REST backend technology	Technology used to deploy REST endpoints.
Backend's location and provider	Location of the servers and the provider for the internet connection.

Client's location and provider	Location of the client and its provider for the internet connection.
Connection speed between the client and the backend	The speed is expressed in Mbps.
REST queries	List the queries that will be executed for the REST part with their number of iterations.
GraphQL queries	List the queries that will be executed for the GraphQL part with their number of iterations.
Execution type	Shows the execution type chosen for the test. It can be either sequential or in parallel.

With all of these characteristics, the reader of the test will have a better understanding of the provenance of the results. As each characteristic has an impact on the result, it helps to correlate those factors with the conclusions. In addition, this table will be used as a template to describe our technical environments for the two situations.

4.1.2 Collecting results

The collecting part has to be done carefully so results can be provided in a precise way from our measurement. To achieve that, we will proceed to the data collection as followed:

1. Copy and store each metric from each result.
2. Make a piece of evidence from our result (screenshot) displaying the date and time of execution.
3. Store pieces of evidence among the copied result if possible.

The multiple storages of data protect us against any data loss or inaccuracy. Moreover, it allows the reader of the report to trace the numbers from the final conclusion to the original measurement.

4.1.3 Compiling results

A single result is not a solid base to bring any conclusions. As results may offer variations from execution to another, the need to perform multiple measurements to get rid of those gaps is cruicial. The more samples you bring in the model, the more accurate your compiled result will be.

Table 2 Example of a results' representation

Result number	Date and time	GraphQL time (ms)	REST time (ms)	GraphQL size (KB)	REST size (KB)
1	2020-05-21, 10:32:35 +02:00	161	303	4.91	6.18
2	2020-05-21, 10:32:41 +02:00	292	357	4.91	6.18
3	2020-05-21, 10:32:46 +02:00	171	334	4.91	6.18
4	2020-05-21, 10:32:49 +02:00	177	319	4.91	6.18
5	2020-05-21, 10:32:51 +02:00	171	326	4.91	6.18
Average	-	194	328	4.91	6.18

This example features five results extracted from a benchmark test made on the specified date in the "Date and time" column. As you can attest, response time offers variations due to uncontrollable factors, such as network traffic or server's load, for example. This is the reason why we compile them inside a single model and exploit the average of all the samples.

4.1.4 Comparing results

Last but crucial, our compiled results will be compared in order to formulate any kind of conclusions. The average output for each metric is easily comparable between the two standards by placing them side by side and see which value is the lowest. In addition to that, the delta between the two technologies with the percentage will be represented. This leads us to formulate a sentence as followed: GraphQL was 48% faster than REST while GraphQL's response is 21% lighter than REST. This sentence makes the result crystal clear even for a non-technical person.

4.2 Situation A

Based on the methodology described previously, the analysis of a concrete case can be started. This first situation features a simple case used by many courses online to illustrate the need for GraphQL.

4.2.1 Technical context

In the first case, we will embody the role of a small start-up which offers a social media application. The company is willing to test the queries for one of their views with GraphQL to identify if they can expect a gain of speed by moving to this new technology. The concerned view displays a post from a specific user and its comments. With the current REST APIs, they need to execute two queries from the front-end to be able to fetch all data.

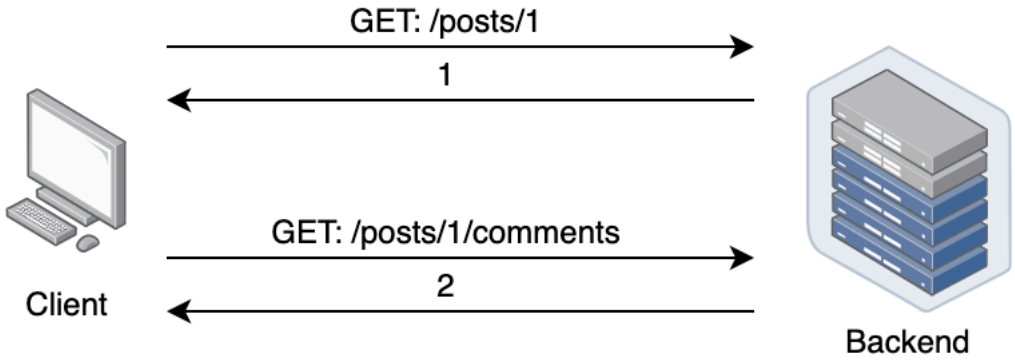


Figure 20 REST queries for situation A

The start-up needs to optimise the fetching process with GraphQL. They aim to wrap this two steps query into a single query. Before having access to the results of this comparison, the technical characteristics from our template defined in the methodology part of our analysis needs to be established.

Table 3 Technical characteristics of situation A

Characteristic	Value
GraphQL backend technology	The backend uses a Strapi based on Node that provides a GraphQL endpoint.
REST backend technology	Strapi mentioned above also provides the REST endpoints.
Backend's location and provider	Google Zürich, Switzerland
Client's location and provider	Swisscom Lausanne, Switzerland

Connection speed between the client and the backend	80Mbps / 27Mbps, 20ms latency with Google Zürich
REST queries	GET: /posts/1 Iterations: 1 GET: /posts/1/comments Iterations: 1

GraphQL queries	{
	posts {
	name
	content
	postDate
	author {
	username
	}
	comments {
	content
	commentDate
	user {
	username
	}
	}
	}
	}
	Iterations: 1

Execution type	Sequential

4.2.2 Results

Based on the given situation, few samples were measured and compiled. They will be presented the same way our methodology defines the presentation of results.

Table 4 Results for situation A

Result number	Date and time	GraphQL time (ms)	REST time (ms)	GraphQL size (KB)	REST size (KB)
1	2020-05-22, 1:05:00 +02:00	184	344	4.91	6.18

2	2020-05-22, 1:05:02 +02:00	166	365	4.91	6.18
3	2020-05-22, 1:05:04 +02:00	187	335	4.91	6.18
4	2020-05-22, 1:05:06 +02:00	164	343	4.91	6.18
5	2020-05-22, 1:05:08 +02:00	210	341	4.91	6.18
6	2020-05-22, 1:05:10 +02:00	169	338	4.91	6.18
7	2020-05-22, 1:05:11 +02:00	206	336	4.91	6.18
8	2020-05-22, 1:05:13 +02:00	191	351	4.91	6.18
9	2020-05-22, 1:05:16 +02:00	201	352	4.91	6.18
10	2020-05-22, 1:05:18 +02:00	195	359	4.91	6.18
Average	-	187	346	4.91	6.18

As you can observe, the gap between the values measured was minimal. For this reason, we decided to limit our sample to ten results as more results would add unnecessary precision. You will find the original results (screenshot) in the appendices part of this thesis.

4.2.3 Conclusions

From this first set of results, we can immediately identify a trend. GraphQL offers faster response time for every sequence of queries. The average delta between the two standards is 159ms, which makes GraphQL 46% faster than REST for the A situation. This result was expected on a theoretical level. As REST needs two times more communications than GraphQL, we could expect a difference of about 50% more execution time for the REST API.

On the other hand, the response size remained stable for every query as the amount of requested information was the same over the whole benchmark process. GraphQL is slightly lighter with its 4.91KB of data. The observed delta is 1.27KB which makes GraphQL 21% lighter than his opponent.

4.3 Situation B

For the second situation, a proportional change of the results will be tried to be identified by increasing the number of REST requests. To achieve that, we will set a new use-case that needs more REST queries while GraphQL will stick to a single query that wraps all the required pieces of information.

4.3.1 Technical context

The same start-up will now try another part of their user interface. In this case, the front-end needs to fetch details about ten followers who have liked a post of an individual user. The main difference with situation A is that requests can be performed in parallel. This leads us to observe the following REST schema.

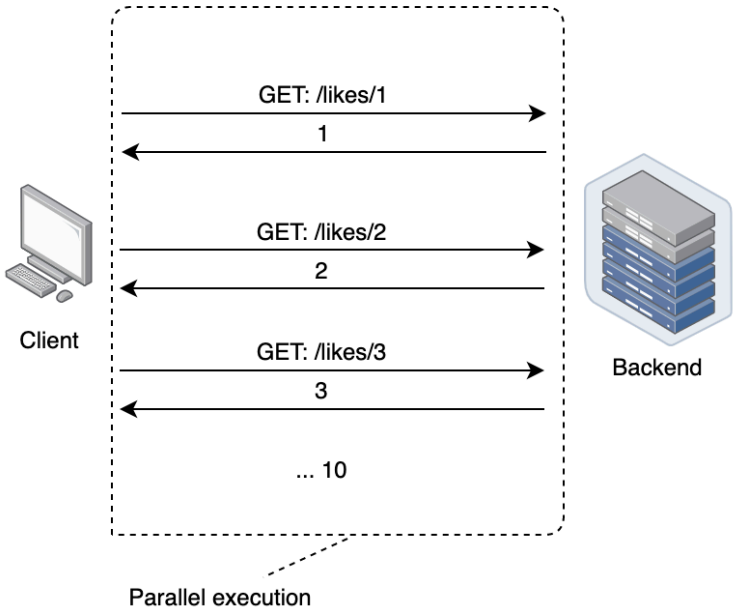


Figure 21 REST queries for situation B

This sequence of queries is considered as a bottleneck for the front-end because the number of likes will define the number of queries. A large number of likes would cause a severe performance issue for the application. In this perspective, GraphQL could help the process by reducing the number of communications to one.

Table 5 Technical characteristics for situation B

Characteristic	Value
GraphQL backend technology	The backend uses a Strapi based on Node that provides a GraphQL endpoint.
REST backend technology	Strapi mentioned above also provides the REST endpoints.
Backend's location and provider	Google Zürich, Switzerland
Client's location and provider	Swisscom Lausanne, Switzerland
Connection speed between the client and the backend	80Mbps / 27Mbps, 20ms latency with Google Zürich
REST queries	GET: /likes/1 Iterations: 1 GET: /likes/2 Iterations: 1 … GET: /likes/10 Iterations: 1

GraphQL queries	{
	likes {
	user {
	username
	email
	}
	}
	}
	Iterations: 1

Execution type	Parallel

4.3.2 Results

With our new test environment setup, the benchmark of our queries from the benchmark tool can start. Data collection will be done, once again, accordingly to our methodology.

Table 6 Results for situation B

Result number	Date and time	GraphQL time (ms)	REST time (ms)	GraphQL size (KB)	REST size (KB)
1	2020-05-22, 4:02:23 +02:00	151	243	1.37	4.74
2	2020-05-22, 4:02:25 +02:00	141	227	1.37	4.74
3	2020-05-22, 4:02:27 +02:00	151	234	1.37	4.74
4	2020-05-22, 4:02:28 +02:00	240	244	1.37	4.74
5	2020-05-22, 4:02:30 +02:00	135	264	1.37	4.74
6	2020-05-22, 4:02:31 +02:00	133	250	1.37	4.74
7	2020-05-22, 4:02:33 +02:00	167	253	1.37	4.74
8	2020-05-22, 4:02:35 +02:00	168	253	1.37	4.74
9	2020-05-22, 4:02:36 +02:00	195	277	1.37	4.74
10	2020-05-22, 4:02:38 +02:00	170	281	1.37	4.74
Average	-	165	253	1.37	4.74

In this second sample, attention will be payed to the fourth result. This result shows a delta of only 4ms compared to the others that offer an average of 88ms. We can explain this singular result with an anomaly caused by the network that slowed down the GraphQL request. This output may have been affected by many factors, so it is hard to explain precisely the source of this particular number.

4.3.3 Conclusions

Despite this anomaly, these results show once again that GraphQL was faster and lighter than REST. Proportionally GraphQL was this time 35% faster, which is 11% less than the situation A. We can explain this small decrease of speed by the type of execution of the requests. With a sequential execution, this test would have taken around three times

longer to execute for the REST queries. Instead of that, the parallel schema allowed REST to be more competitive with GraphQL.

Taking a look at the response size, this time, GraphQL was significantly lighter. The reason is the quantity of unnecessary data that comes with the high number of communications made by REST. Because of this factor, GraphQL stands out by being 71% lighter than REST.

5 Discussions

To conclude our research thesis, we necessarily need to attempt giving an answer to our preliminary research question. Additionally, the methods used during our research will be discussed. Finally, I will give my own perspective on the technology and my personal findings.

5.1 Answer to the research question

This research thesis started in the perspective of answering a question that has never been answered with a quantitative approach. As a reminder here is the question that this thesis is attempting to clarify: Is GraphQL faster and better optimised than REST?

Based on our methodology we can attest that the answer is positive for the situation A and B. On both cases we found results that were giving the advantage to GraphQL as it was 35% to 46% faster and 21% to 71% lighter on average. As discussed in the delimitations of research topic, these conclusions are only valid with our methodology applied for the two situations exposed. Other situations may give different results that may also change the positive affirmation to the research question.

To sum up, it would not be relevant to take this answer as a definitive statement about those two technologies. On the market there are an infinity of cases that can be tested with our methodology, but because of time and budget limitations, it is the reader's responsibility to consider testing his own situation to have a final answer.

5.2 Methodology discussion

Besides the central goal of answering a research question, this thesis also had the purpose of proposing a methodology to compare two technologies side by side. This paper brought the reader the ability to reproduce our findings as much as creating his own situations to collect dedicated results.

Overall, that this part can be considered as a success. Our methodologies are transparent and detailed at every step. Any additional tool or technology used to collect and analyse the results are accessible for free online. In addition, the methodology respected the quantitative approach by sticking to figures evaluation and factual reasoning.

5.3 Personal conclusion

From a personal perspective, this research thesis taught me how to structure analysis in order to give a non-biased judgment by using a quantitative approach. This type of question answering is a considerable skill that will help my clients and me to make the best decisions based on valid and proven arguments.

Regarding the technology itself, I have never doubted that GraphQL is a big step forward in term of query standard. From a developer angle, GraphQL addresses many issues that traditional REST APIs were hanging behind for years. When it comes to choosing GraphQL over REST, there are plenty of other factors to look at than just performances criteria. As a result, I truly think that this methodology is important to be used among other types of researches and findings.

References

Wikipedia. (2020, May 1). *Representational state transfer*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Representational_state_transfer

Fielding, R. T. (2000). *Fielding Dissertation.* Retrieved from ics.uci.edu: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Wodehouse, C. (2017, 04 19). *SOAP vs. REST: A Look at Two Different API Styles.* Retrieved from UpWork: https://www.upwork.com/hiring/development/soap-vs-rest-comparing-two-apis/

Wikipedia. (2020, 05 2). *ACID.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/ACID

Novarum & PCWorld. (2012, 04 16). *3G and 4G Wireless Speed Showdown: Which Networks Are Fastest?* Retrieved from PCWorld: https://www.pcworld.com/article/253808/3g_and_4g_wireless_speed_showdown_which_networks_are_fastest_.html

Byron, L. (2019, 03 15). *Keynote: A Brief History of GraphQL - Lee Byron, Co-Creator of GraphQL.* Retrieved from Youtube: https://www.youtube.com/watch?v=VjHWkBr3tjI

How to GraphQL. (2020, 05 03). *Big Picture (Architecture).* Retrieved from How to GraphQL: https://www.howtographql.com/basics/3-big-picture/

Jung, D. (2019, 9 7). *Using RESTful APIs versus GraphQL.* Retrieved from Medium: https://medium.com/swlh/using-restful-apis-versus-graphql-1e6c350d56c9

Aiyer, A. (2018, 02 17). *Exploring different GraphQL Clients.* Retrieved from Medium: https://medium.com/open-graphql/exploring-different-graphql-clients-d1bc69de305f

Cerroni, E. (2019, 01 24). *How to update the Apollo Client's cache after a mutation.* Retrieved from Medium: https://medium.com/free-code-camp/how-to-update-the-apollo-clients-cache-after-a-mutation-79a0df79b840

GraphQL Foundation. (2020, 05 5). *Schemas and Types.* Retrieved from GraphQL documentation: https://graphql.org/learn/schema/

Prisma. (2017, 07 17). *HowToGraphQL (Fundamentals) - Core Concepts (3/4).* Retrieved from Youtube: https://www.youtube.com/watch?v=NeQfq0U5LnI

GraphQL Foundation. (2020, 05 7). *Code.* Retrieved from GraphQL documentation: https://graphql.org/code/#java-android

Apollo GraphQL. (2020, 05 7). *Schema basics.* Retrieved from Apollo Documentation: https://www.apollographql.com/docs/apollo-server/schema/schema/#the-query-type

Apollo GraphQL. (2020, 05 08). *Resolvers.* Retrieved from Apollo Documentation: https://www.apollographql.com/docs/apollo-server/data/resolvers/

Rhyne, A. E. (2017, 03 28). *GraphQL Tutorial — Getting Started.* Retrieved from Medium: https://medium.com/@thebigredgeek/graphql-tutorial-getting-started-f97c8e03156e

Wikipedia. (2020, 05 9). *React (web framework).* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/React_(web_framework)

Wikipedia. (2020, 05 10). *React Native.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/React_Native

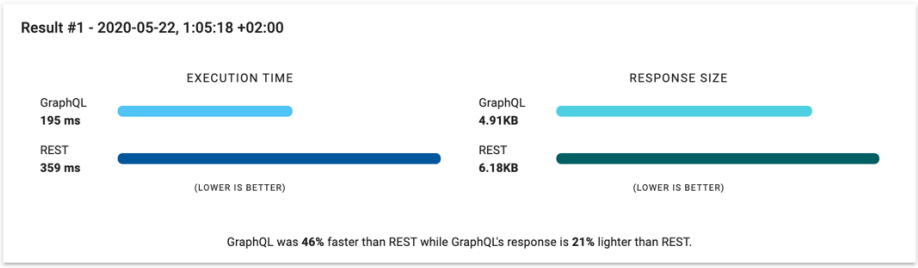
Shakya, N. (2018, 08 28). *5 super reasons to love React JS.* Retrieved from Medium: https://blog.usejournal.com/5-super-reasons-to-love-react-js-b57505535ea5

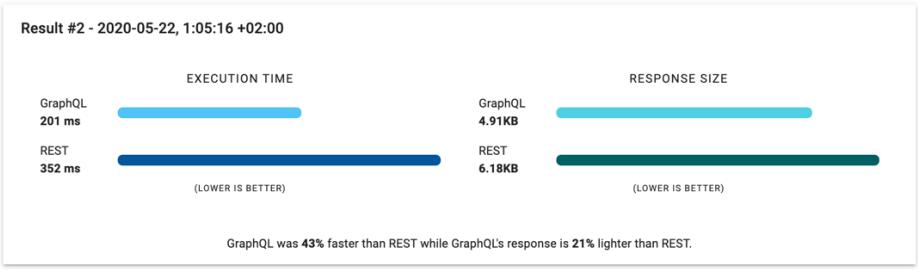
Wikipedia. (2020, 05 12). *GitHub*. Retrieved 05 2020, from Wikipedia: https://en.wikipedia.org/wiki/GitHub

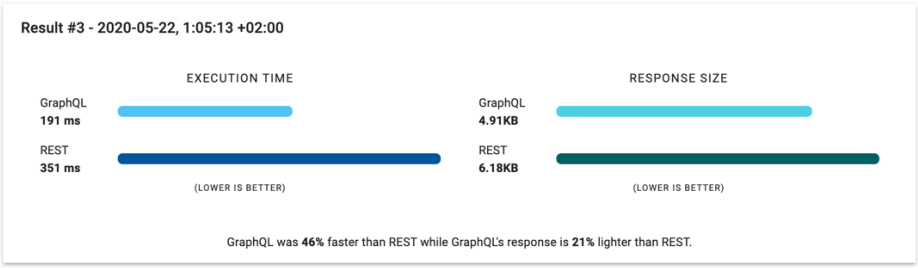
Wikipedia. (2020, 05 16). *Quantitative research*. Retrieved 05 2020, from Wikipedia: https://en.wikipedia.org/wiki/Quantitative_research

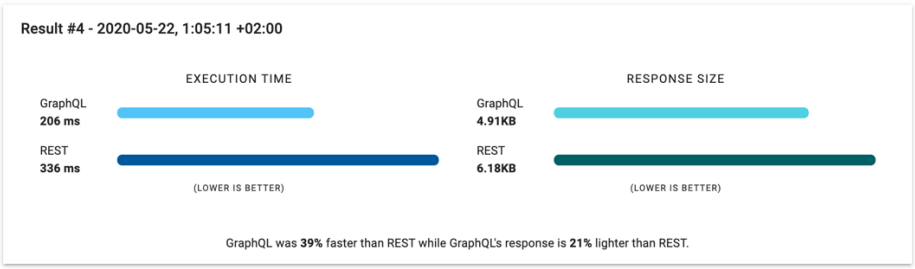
Appendices

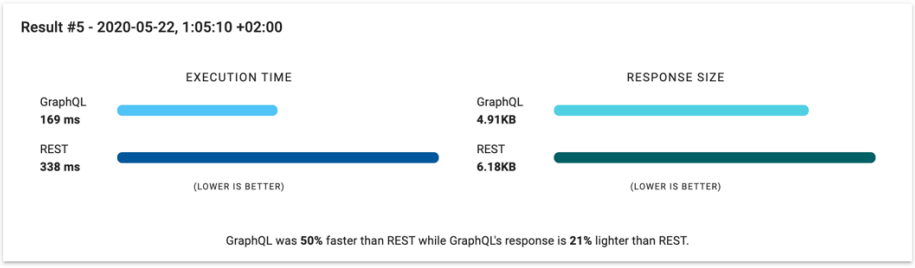
Appendix 1. Original results for solution A

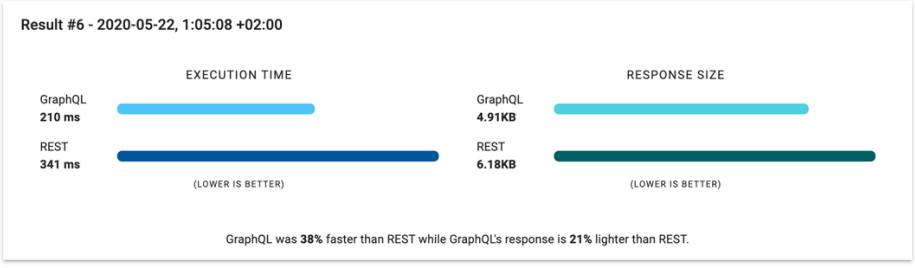




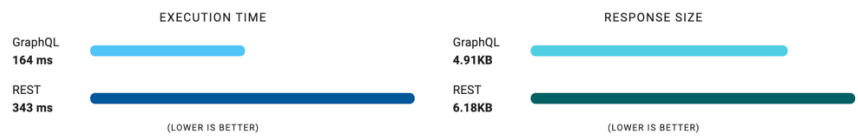






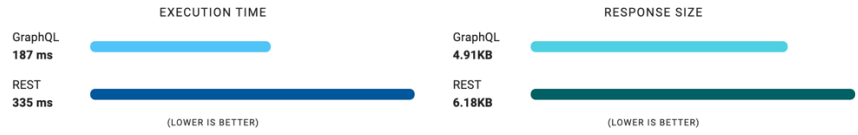


Result #7 - 2020-05-22, 1:05:06 +02:00



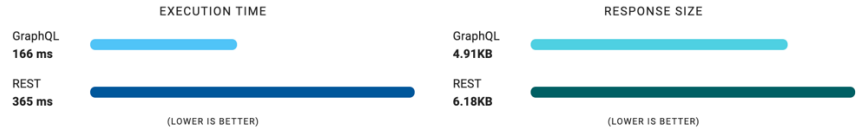
GraphQL was **52%** faster than REST while GraphQL's response is **21%** lighter than REST.

Result #8 - 2020-05-22, 1:05:04 +02:00



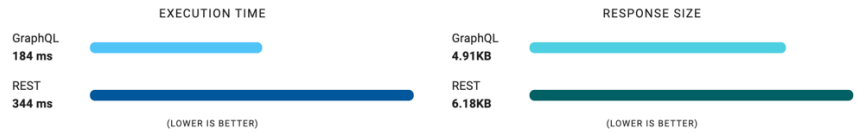
GraphQL was **44%** faster than REST while GraphQL's response is **21%** lighter than REST.

Result #9 - 2020-05-22, 1:05:02 +02:00



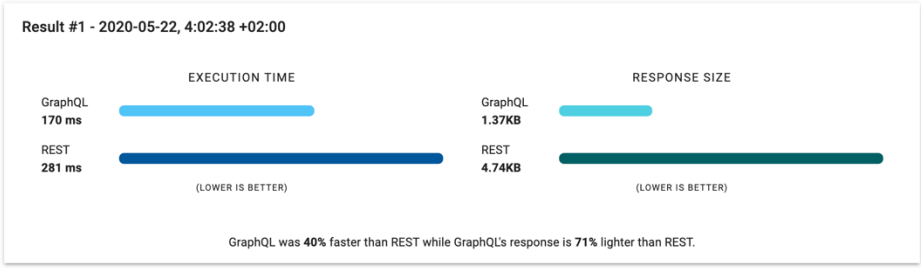
GraphQL was **55%** faster than REST while GraphQL's response is **21%** lighter than REST.

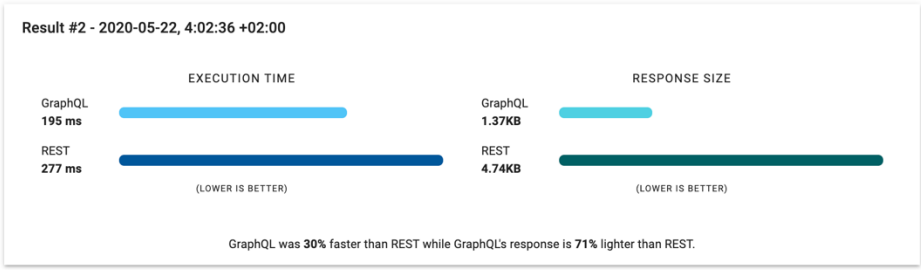
Result #10 - 2020-05-22, 1:05:00 +02:00

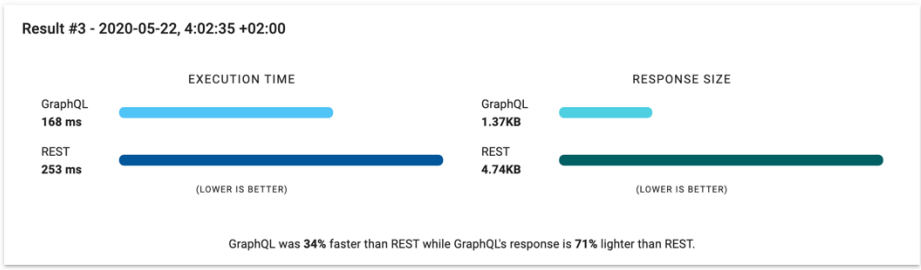


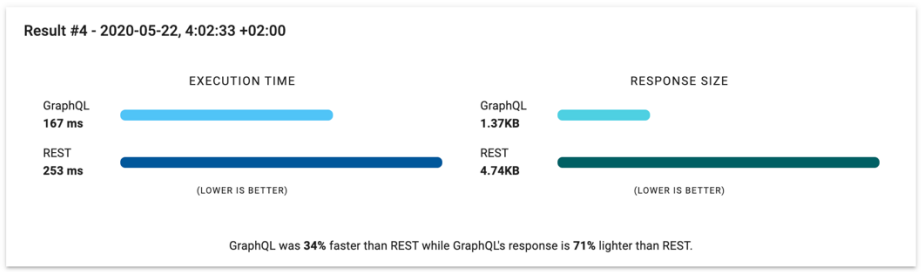
GraphQL was **47%** faster than REST while GraphQL's response is **21%** lighter than REST.

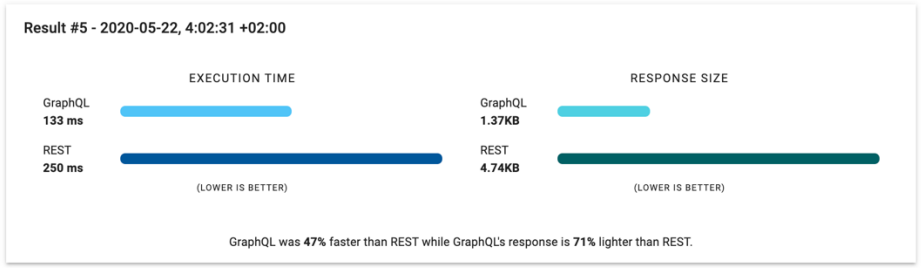
Appendix 2. Original results for solution B

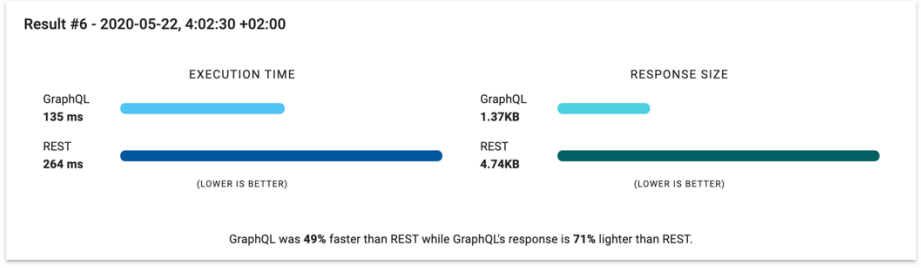




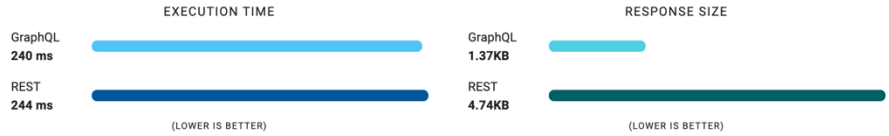






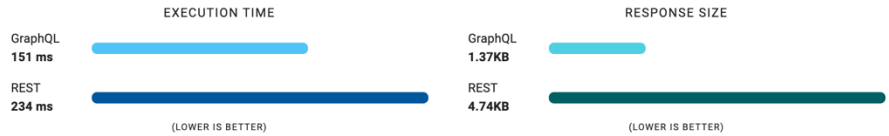


Result #7 - 2020-05-22, 4:02:28 +02:00



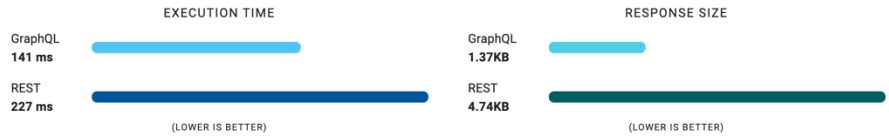
GraphQL was **2%** faster than REST while GraphQL's response is **71%** lighter than REST.

Result #8 - 2020-05-22, 4:02:27 +02:00



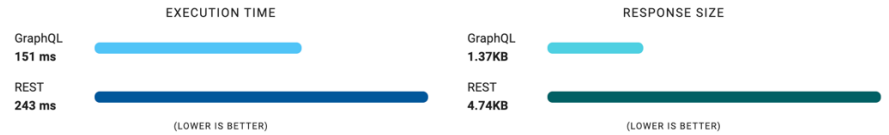
GraphQL was **36%** faster than REST while GraphQL's response is **71%** lighter than REST.

Result #9 - 2020-05-22, 4:02:25 +02:00



GraphQL was **38%** faster than REST while GraphQL's response is **71%** lighter than REST.

Result #10 - 2020-05-22, 4:02:23 +02:00



GraphQL was **38%** faster than REST while GraphQL's response is **71%** lighter than REST.