International Journal of Computer Applications (0975 - 8887)
Volume 184 - No.20, July 2022

Enhancement of GraphQL Data Fetching Technique

Chaitanya A.
Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering
Chennai,India

Hariharan R.
Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering
Chennai,India

Harishkumaar S.
Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering
Chennai,India

Prabavathy B.
Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering
Chennai,India

ABSTRACT

Data fetching techniques are used to communicate between servers and clients on the World Wide Web. It serves as the basis of modern data-driven websites, since these websites have more frequent communication between client and server. In addition, it will have a huge data transfer between them. The predominant data fetching techniques are REST API, GraphQL and Protobuf. REST API is the most popular data fetching technique in the market, but it does not provide a way to get only the fields required by the client. GraphQL addresses this problem by providing dynamic querying. However, both the REST API and GraphQL return the response in JSON format, which has keys and values. These keys in the response can occupy a huge chunk of the response as the size of the response increases. This is a drawback for data-driven applications that run on bandwidth-constrained devices. Protobuf tries to address this problem by returning the response as an array of values, but Protobuf does not allow the client to query only the required fields. Hence, the objective of this paper is to propose a new data fetching technique that combines the advantages of GraphQL and Protobuf to reduce the payload size and the response time.

General Terms

API, RESTful, GraphQL, Protobuf, E-GraphQL, Schema Definition Language , Abstract Syntax Tree , HTTP, Query

Keywords

Payload size, Response time

1. INTRODUCTION

Data fetching techniques play an important role in modern web applications, which essentially means fetching data from the server by sending requests. In modern web applications, many websites are data-driven, like Instagram, Twitter, Amazon and Facebook, which make requests to the server and fetch data frequently. Though there are several data fetching techniques, the most prevalent methods are REST API, GraphQL and Protobuf. These techniques are not suitable for all use cases and they have their own limitations, like over-fetching, under-fetching and incapability of caching.

1.1 Representational State Transfer (REST)

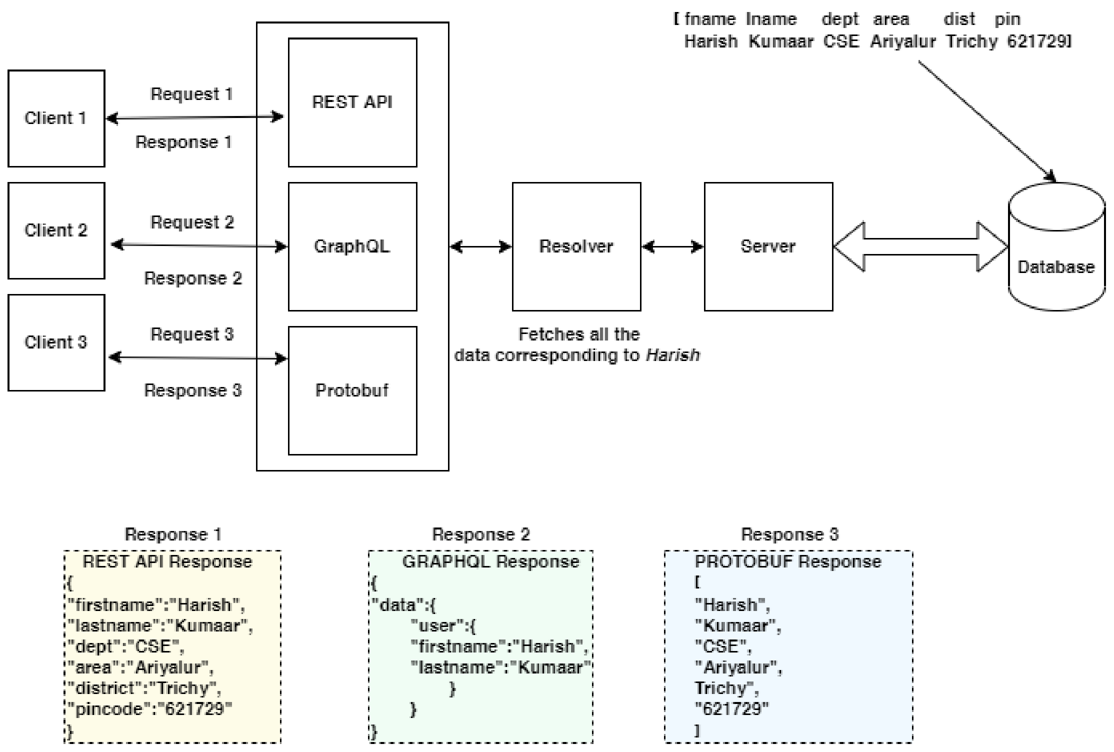
REST stands for Representational State Transfer. It's a software architectural style for implementing web services. In REST APIs, data is exposed by means of endpoints. Each endpoint returns data about one resource, and each resource has a predefined set of fields. REST APIs communicate via HTTP methods like GET, POST, PUT and DELETE. This REST architectural style defines six guiding constraints called client-server architecture, statelessness, cacheability, layered systems and code-on-demand.

1.2 GraphQL

GraphQL is a query language for the API. In GraphQL, data is exposed as a graph, defined by means of a schema. Each node of this schema represents objects and contains fields. Each field has a name and a type. Edges appear when a field references another object. Clients access a GraphQL service through a single endpoint, which is used to submit queries and mutate data. It provides a domain-specific language called Schema Definition Language (SDL) for defining schemas, including types and queries.

1.3 Protocol Buffers

Protobuf is language agnostic and provides a way to serialize and deserialize data. The schema in Protobuf is written in the proto language and then compiled into other languages. The languages supported by protobuf are Kotlin, Dart, Go, Ruby and C. In Protobuf, the schema of the message to be sent from the server to the client is defined in a .proto file. The fields in the protobuf structure are assigned field numbers so that they can be sorted into an array while sending data to the client. The field number must be unique. The written proto files are compiled into any language to be used natively in that language. The data from the server is serialized before being sent to the client using the classes generated by compiling the



REST API Response
```
{
"firstname":"Harish",
"lastname":"Kumaar",
"dept":"CSE",
"area":"Ariyalur",
"district":"Trichy",
"pincode":"621729"
}
```

GRAPHQL Response
```
{
"data":{
    "user":{
    "firstname":"Harish",
    "lastname":"Kumaar"
        }
    }
}
```

PROTOBUF Response
```
[
"Harish",
"Kumaar",
"CSE",
"Ariyalur",
Trichy",
"621729"
]
```

Fig. 1. Request-Response Generation in various data fetching techniques

.proto files. The serialized data from the server is then deserialized using the classes generated by compiling the .proto files, and the data is retrieved.

1.4 Request Response generation by various data fetching techniques

In the Fig 1 the server maintains the personal information related to the users. The personal information of a user consists of *first name, last name, department, area, district* and *pincode*. A sample query is made to retrieve the first and last name of a given user. The REST API server returns the entire data to the client in JSON format without any filters that makes REST API to consume larger space and time[2]. GraphQL provides only the necessary information as key-value pairs, however the keys in the response use more bandwidth. The Protobuf response contains only data without keys in a sorted array format, however it has the issue of dynamic querying.

GraphQL's key benefit over the REST API is that it prevents underfetching and over-fetching. This addresses the REST API's limitations, but the main issue is JSON itself. When a response is sent to the client from the server, the response contains both keys and their respective values. The keys in the JSON response payload take up a significant amount of size in the response payload. These keys can be removed from the payload in order to reduce the payload's size. Hence, a new approach to data fetching, Enhanced GraphQL (E-GraphQL), is proposed. E-GraphQL takes the GraphQL approach of limiting under-fetching and over-fetching and applies it to the Protobuf approach of maintaining the schema of the data locally on both the server and client sides to limit payload size.

The rest of this paper is organized as follows: Section 2 discusses in detail about the existing work related to data fetching techniques; Section 3 describes the proposed system architecture and the modules; Section 4 discusses the evaluation of the proposed system; Section 5 discusses the conclusion and future work.

2. RELATED WORK

Several research work focused mainly on the comparative analysis of different data fetching techniques based on different parameters like caching, response time.

Andersson et al. [1] have studied two data fetching techniques, namely REST API and GraphQL in order to compare them with caching. An application has been developed using the REST API and GraphQL. This application has been tested for response time by disabling and enabling caching. From the experimental results, the response time for the REST API is better when compared to GraphQL.

Brito et al. [2] proposed that, when two data fetching techniques, namely REST API and GraphQL be compared for easier and faster development. An application has been developed using REST and GraphQL by 22 students. This application has been tested for the implementation time by the developers. From the experimental results, GraphQL outperforms REST even among more experienced participants. It also does the same among the participants with previous experience of REST, but not with GraphQL.

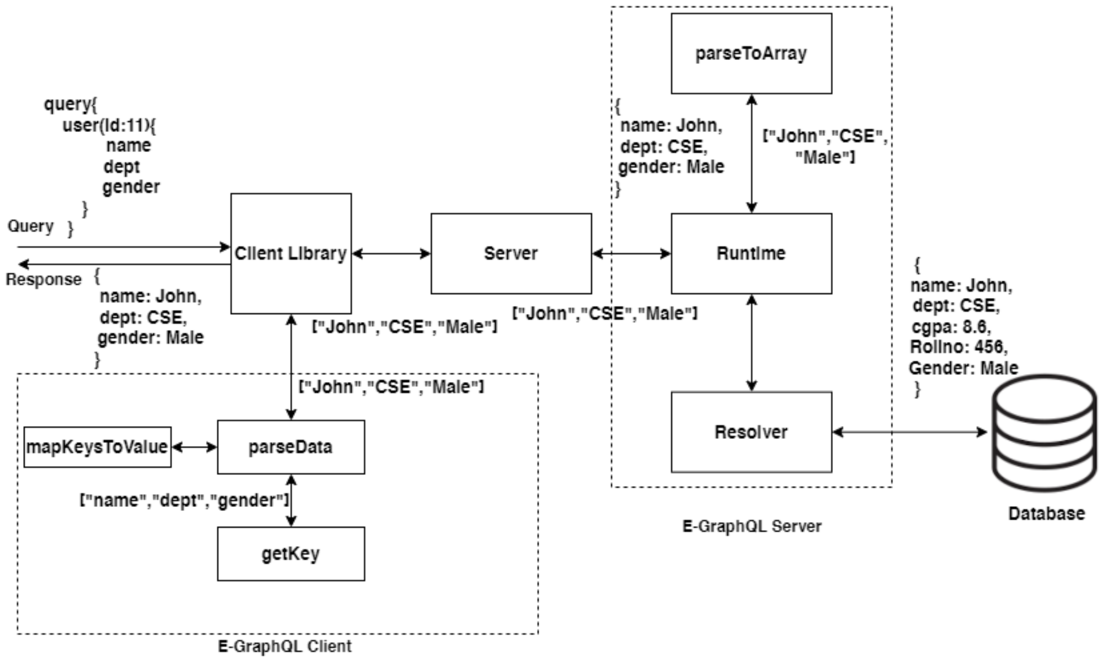


Fig. 2. Proposed design of E-GraphQL data fetching technique

Seabra et al. [7] experimented with various architectural models and came up with an optimized one. To infer the performance issues, two target applications were developed using two web service architectures REST API and GraphQL. The response time and the average transfer rate between the requests have been analyzed. It was noticed that after migration, GraphQL performed below its REST counterpart for workloads above 3000 requests, ranging from 98 to 2159 Kbytes per second. It was also observed that migrating to GraphQL resulted in an increase in performance by two-thirds of the tested applications, with respect to the average number of requests per second and transfer rate of data.

Sumaray et al. [8] have compared four different data serialization formats with an emphasis on serialization speed, data size, and usability. The selected serialization formats include XML, JSON, Thrift, and ProtoBuf. XML and JSON are the most well known text-based data formats, while ProtoBuf and Thrift are relatively new binary serialization formats. These data serialization formats are tested on an Android device using both text-heavy and number-heavy objects.

3. PROPOSED DESIGN OF E-GRAPHQL

E-GraphQL is a new hybrid API technique that is obtained by combining the core principles of GraphQL and Protocol Buffers. The proposed E-GraphQL data fetching technique, has two modules namely E-GraphQL server and E-GraphQL client modules. This section describes in detail the E-GraphQL technique with an use case.

3.1 E-GraphQL Server Module

Initially, a GraphQL query for the user with *id:11* requesting name, dept, gender is sent to the *client-library* as shown in Fig 2. The *client-library* saves the query and sends the request to the server, which is further forwarded to the GraphQL runtime. It invokes the respective query resolver. Here, the resolver is responsible to retrieve the data of user *id:11* which consists of *name, dept, cgpa, Rollno, Gender* from the data source and sends it to the runtime. The output of resolver, which is of JSON format where all the data of the user is filtered based on the query that is only *name, dept, gender*. Then the runtime calls the *parse-to-array* function which converts the JSON object to array of values by removing all the keys present and returns the array of values which is *[”John”, ”CSE”, ”Male”]* to server.

3.2 E-GraphQL Client Module

On receiving the output from the server, client-library sends it to the *parse-data* function along with the saved query. These two inputs are passed to *get-keys* function. This function will extract only the relevant keys from the saved query. Extracted keys and the output from the server are passed to the *map-key-value* function, which maps the list of keys and the output from the server to a JSON object. This JSON object is returned to the *parse-data* function, which further sends it to the client-library as shown in Fig 2. This final object represents the intended response used in the application.

4. SYSTEM EVALUATION

This section discusses in detail about the implementation, performance metrics and experimental results of the proposed system.

4.1 Implementation

The GraphQL Foundation has open sourced the implementations of its runtime in many programming languages. The proposed technique was using Javascript implementation of GraphQL runtime. The *index.js* file of *express-graphql* receives the query sent to the server, then passes it to the *graphql library*. The response returned

from the GraphQL runtime is in JSON format and is sorted in the order of the fields based on the query.
A new *parse-to-array* functionality has been introduced in E-GraphQL which takes the response from the GraphQL runtime and removes all the keys in the JSON data, which is discussed below.

—If the response is an array, a special indicator is added to the array, which later helps the frontend to parse the response.
—Each element of the array is traversed and passed back to the *parse-to-array* function. If the response is an object the keys are removed and the values are added to an array.
—Each element of the array/object is traversed and processed as mentioned above, in order to generate the final response which is pure without any keys. This parsed array response is sent to the client.

The client library for E-GraphQL has been developed from scratch. It has a main function called *get*, which accepts two inputs called the endpoint of the E-GraphQL server and the GraphQL query to be sent to the server. The *get* function receives a response which is an array of values, and further passes to the *parseData* function, which is written to parse the received data back to JSON format by following steps.

—If the type of element is either boolean or int or string, the key is maintained without changes.
—If the type of the element is an object, then a key that corresponds to that object is added along with a special character as the prefix to all the keys inside the object and the object is iterated again.
—If the type of the element is an array, the length of the array is obtained and the keys are duplicated as per the length of the array.

The *parse-data* function then takes the array of keys and the array of values and passes them to the *map-key-value* function. In *map-key-value* function iterates through the array of keys and the array of values. Each key and value is added to a result object and returned to the application.

4.2 Performance Metrics

In order to test the performance of GraphQL and E-GraphQL data fetching techniques, a real time data-driven application, namely the Instagram application has been cloned. These two techniques were tested with the performance measures, namely *payload size* and *response time*.

Payload size is an important metric used to measure the performance of any application. Applications with less payload sizes tend to be perform better as they load faster when compared to the applications that have higher payload sizes.

Response time is the time taken for the client to receive the response from the server. An application that has a very less response time is more performant because it attracts more users than applications that have higher response time.

4.3 Experimental Results

Multiple experiments have been conducted to test the performance of the proposed E-GraphQL technique with the necessary performance metrics, such as data loss, payload size and response time. Since E-GraphQL makes use of HTTP, which is based on TCP, there will not be loss of information during data transfer. This is due to the fact that TCP is a connection-oriented protocol and is designed to carry packets across networks and ensure that data is delivered correctly.

4.3.1 Impact of E-GraphQL on payload size.
Objective: To determine the payload size for the GraphQL and E-GraphQL.

Table 1. Analysis of payload size (in KB) between GraphQL and E-GraphQL

No. of	**GraphQL**	**E-GraphQL**
Posts	**Payload size in KB**	**Payload size in KB**
100	26.69	21.79
200	53.35	43.58
300	79.91	65.25
400	106.4	86.85
500	133.04	108.62
600	160	130
700	185.52	151.33
800	211.78	172.7
900	238.09	194.14
1000	264.46	215.63

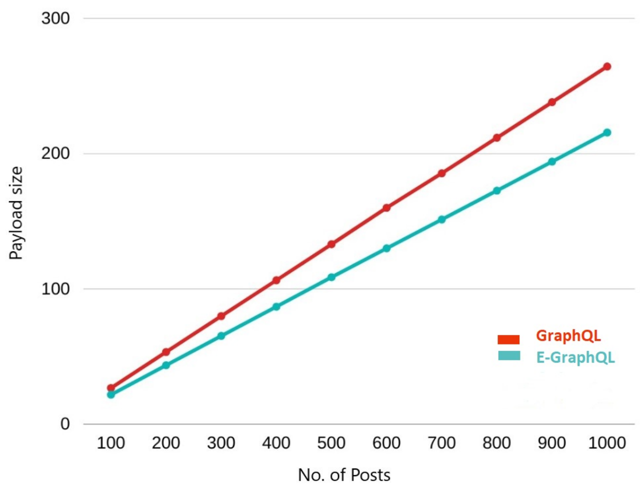


Fig. 3. Payload size representation of GraphQL and E-GraphQL

The *size of the payload* is measured by using either the developer tools offered by Google Chrome browser or Postman tool. When making a request to the server, the *size of payload* is mentioned in the network tab of Google Chrome browser and also in Postman.

Queries for a *number of posts* were made to both GraphQL and E-GraphQL servers and a series of readings were recorded. The recordings are tabulated as shown in the Table 1. A graph has been plotted to visualize the recordings.

From the graph it is inferred that there is a steady increase in payload size and difference between the payload sizes of GraphQL and E-GraphQL as the number of posts increases as shown in Figure 3. This shows that E-GraphQL decreases the payload size significantly when compared with GraphQL. The average difference between the payload sizes of GraphQL and E-GraphQL is 26.935 KB. This proves that E-GraphQL reduces the payload size for any input size compared to GraphQL.

4.3.2 Impact of E-GraphQL on response time.
Objective: To determine the response time for the GraphQL and E-GraphQL.

Table 2. Analysis of response time (in ms) between GraphQL and E-GraphQL

No. of	**GraphQL**	**E-GraphQL**
Posts	**Response time in ms**	**Response time in ms**
100	135	110
200	150	137
300	170	158
400	187	173
500	196	183
600	211	200
700	245	213
800	265	263
900	300	268
1000	383	346

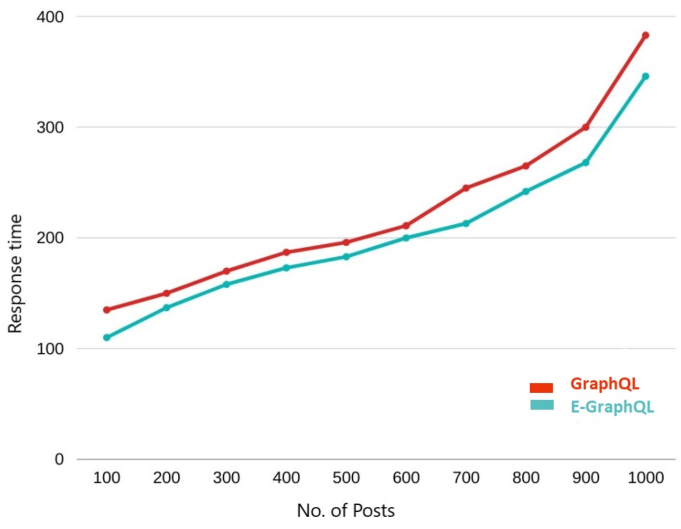


Fig. 4. Response time representation of GraphQL and E-GraphQL

The *response time* is also measured by using either the developer tools offered by Google Chrome browser or Postman tool. While receiving the response from the server, the *response time* is recorded in the network tab of Google Chrome browser and also in Postman.
Queries for a *number of posts* were made to both GraphQL and E-GraphQL servers and a series of readings were recorded. The recordings are tabulated as shown in the Table 2. A graph has been plotted to visualize the recordings. From the graph it is inferred that there is a steady increase in response time and in the difference between the response time of GraphQL and E-GraphQL as the number of posts increases as shown in Figure 4. This shows that E-GraphQL decreases the response times significantly when compared with GraphQL. The average difference between the response time of GraphQL and E-GraphQL is 19.1 ms. This proves that E-GraphQL reduces the reponse time for any input size compared to GraphQL.

From the above figures, it is inferred that the *payload size* in E-GraphQL is reduced when compared with GraphQL. With a lower *payload size*, it is possible to use E-GraphQL on *bandwidth-constrained* devices.

5. CONCLUSIONS AND FUTURE WORK

Data fetching techniques should be more efficient for data driven applications that are accessed from bandwidth constrained devices. In this context, a new data fetching technique named E-GraphQL has been proposed. It has been built upon the existing GraphQL runtime. The server module of E-GraphQL has been built using an open-source *express-graphql*. The E-GraphQL client module has been developed from the scratch. The proposed E-GraphQL server and client were published as libraries in Node Package Manager (NPM).

The performance of the proposed E-GraphQL has been tested with metrics like payload size and response time. It is inferred from the experimental results that the proposed E-GraphQL data fetching technique reduces the payload size by an average of 20%. E-GraphQL reduces the response time by an average of 10%. This shows that E-GraphQL is more efficient for data driven applications that run on bandwidth constrained devices.

In future, the open sourced E-GraphQL server and client libraries will be updated regularly as the GraphQL runtime gets updated. The new suggestions from the open source community will be added to the libraries to improve the standards and performance of the libraries.

6. REFERENCES

[1] Andersson, T., Reinholdsson, H., (2021). “REST API vs GraphQL - A literature and experimental study”, Spring Semester, Kristianstad University, Sweden.

[2] Brito, G., Valente, MT., (2020). “Rest vs graphql: A controlled experiment”, In Proceedings of IEEE International Conference on Software Architecture, pp. 81-91.

[3] Google Developers, (2021), Language Guide — Protocol Buffers — Google Developers, https://developers.google.com/protocol-buffers/docs/overview, Accessed on 02/06/2022.

[4] GraphQL Foundation, (2021), GraphQL - A Query language for your API, https://graphql.org, Accessed on 02/06/2022.

[5] How does graphQL work, anyway? - Medium article - https://medium.com/@rajeshdavid/how-does-a-graphql-service-work-internally-496dc9264096, Accessed on 02/06/2022.

[6] IBM Education, (2021), What is a REST API?, https://www.ibm.com/in-en/cloud/learn/rest-apis, Accessed on 02/06/2022.

[7] Seabra, M., Nazário, MF., Pinto, G., (2019). “REST or GraphQL? A performance comparative study”, In Proceedings of the XIII Brazilian Symposium on Software Components, Architectures and Reuse, pp. 123-132.

[8] Sumaray, A., Kami Makki S., (2012). "A comparison of data serialization formats for optimal efficiency on a mobile platform", In Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, pp. 1-6.