Semantics and Complexity of GraphQL

Olaf Hartig
Dept. of Computer and Information Science (IDA), Linköping University
olaf.hartig@liu.se

Jorge Pérez
Department of Computer Science, Universidad de Chile
Millenium Institute for Foundational Research on Data
jperez@dcc.uchile.cl

ABSTRACT

GraphQL is a recently proposed, and increasingly adopted, conceptual framework for providing a new type of data access interface on the Web. The framework includes a new graph query language whose semantics has been specified informally only. This has prevented the formal study of the main properties of the language.

We embark on the formalization and study of GraphQL. To this end, we first formalize the semantics of GraphQL queries based on a labeled-graph data model. Thereafter, we analyze the language and show that it admits really efficient evaluation methods. In particular, we prove that the complexity of the GraphQL evaluation problem is NL-complete. Moreover, we show that the enumeration problem can be solved with constant delay. This implies that a server can answer a GraphQL query and send the response byte-by-byte while spending just a constant amount of time between every byte sent.

Despite these positive results, we prove that the size of a GraphQL response might be prohibitively large for an internet scenario. We present experiments showing that current practical implementations suffer from this issue. We provide a solution to cope with this problem by showing that the total size of a GraphQL response can be computed in polynomial time. Our results on polynomial-time size computation plus the constant-delay enumeration can help developers to provide more robust GraphQL interfaces on the Web.

ACM Reference Format:
Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186014

1 INTRODUCTION

After developing and using it internally for three years, in 2016, Facebook released a specification [5] and a reference implementation of its GraphQL framework. This framework introduces a new type of Web-based data access interfaces that presents an alternative to the notion of REST-based interfaces [16]. One of its main advantages is its ability to define precisely the data you want, replacing multiple REST requests with a single call [5, 6]. Since its release, GraphQL has gained significant momentum and has been adopted by an increasing number of users including Coursera, Github, Neo4J, and Pinterest [9]. A core component of the GraphQL framework is a query language for expressing the data retrieval requests issued to GraphQL-aware Web servers. While there already exist a number of implementations of this language, a more fundamental understanding of the properties of the language is missing. The goal of this paper is to close this gap, which is a fundamental step to clarify intrinsic limitations and, more importantly, to identify optimization opportunities of possible implementations.

To illustrate some of these limitations and optimization opportunities, consider the public GraphQL interface provided by Github [6]. Figure 1(a) shows a query over this interface and Figure 1(b) illustrates the corresponding query result.[1] This query retrieves the login names of the owners of the first two Github repositories that are listed for the user with login "danbri" (which happens to be "danbri" himself in both cases[2]). As our experiments with this public GraphQL interface show, there is an intriguing issue with the size of a query result when we begin nesting queries. Assume that we extend our example into some kind of *path expressions* that discover repository owners by traversing the relationships between Github repositories and their owners in increasing levels of distance. Figure 1(a) represents the level-1 version of such a traversal. The level-2 version, illustrated in Figure 1(c), retrieves the owners of the (first two) repositories that are listed for each repository owner in the result of the level-1 version, and so on. Figure 1(d) shows that there is an exponential increase of the result sizes for levels 1–7. We note that this issue is somehow acknowledged by the Github GraphQL interface and, as a safety measure to avoid queries that might turn out to be too resource-intensive, it introduces a few syntactic restrictions [7]. As one such restriction, Github imposes a maximum level of nesting for queries that it accepts for execution.

However, even with this restriction (and other syntactic restrictions imposed by the Github GraphQL interface [7]), Github fails to avoid all queries that hit some resource limits when executed. For instance, when we replace `first:2` by `first:5` in the queries of our experiment, we observe not only exponential behavior of result size growth and query execution times (cf. Figure 1(e)), but we also receive timeout errors for the level-6 and level-7 versions of the queries. The response messages with these timeout errors arrive from the server a bit more than 10 seconds after issuing the requests. Hence, Github's GraphQL processor clearly tries to execute these queries before their execution times exceed a threshold. Developers have already embarked trying to cope with this and similar issues [1, 20] defining ad hoc notions of "complexity" or "cost" of GraphQL queries. As we explain in this paper these approaches fall short on providing a robust solution for the problem as they can fail in both directions: discarding requests in which an efficient evaluation is possible, and allowing requests in which a complete evaluation is too resource intensive.

Instead of trying to tackle these and other issues by ad hoc solutions, we propose to study them from a formal point of view

ACM ISBN 978-1-4503-5639-8/18/04.
https://doi.org/10.1145/3178876.3186014

[1] All the query executions on which we report have been performed on Oct. 3, 2017.
[2] When increasing the number of repositories to be considered, by changing `first:2` to, say, `first:10`, we also find repositories with other owners.

```
query {
  user(login: "danbri") {
    repositories(first: 2) {
      nodes {
        owner {
          login
} } } } }
```

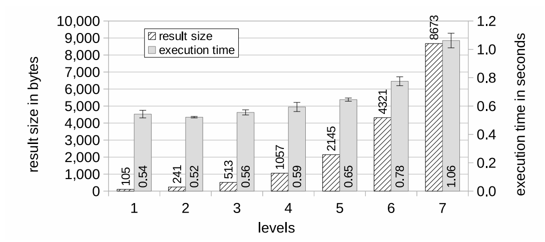
(a) initial example query

```
data: {
  user: {
    repositories: {
      nodes: [
        { owner: { login: "danbri" } },
        { owner: { login: "danbri" } }
      ]
} } }
```

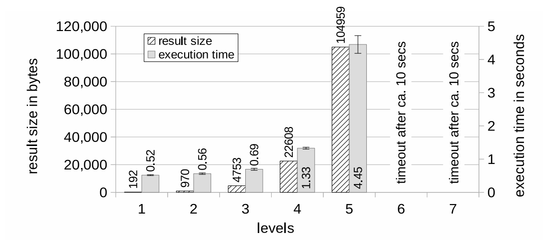
(b) result of initial example query

```
query {
  user(login: "danbri") {
    repositories(first: 2) {
      nodes { owner {
        repositories(first: 2) {
          nodes { owner {
            login
} } } } } } } }
```

(c) level-2 version of the example query



(d) measurements for the example queries with `first:2`



(e) measurements for the example queries with `first:5`

Figure 1: GraphQL queries and responses over the Github GraphQL interface [6]

borrowing the long tradition and tools used by the database community to study the semantics and complexity of query languages. This paper is a first step in this direction. Our formalization and technical results allow us, among other things, to provide a robust solution for the above mentioned problems.

The semantics of GraphQL queries—i.e., the definition of what the expected result of any given query is—is given in the GraphQL specification by means of a recursive program specified by pseudo code. This recursion is based on an operation to resolve any so-called "field" in a query (such as `user`, `repositories`, and `owner` in our example query). Surprisingly, this operation is not fully specified and, instead, simply assumes access to an *"internal function [...] for determining the [...] value of [the] field"* [5]. While the lack of a more precise definition of this internal function may be intentional (to allow for implementations of GraphQL on top of arbitrary database back-ends), it makes a systematic analysis of the GraphQL language unworkable. That is, without a complete formal definition it is impossible to determine properties such as the expressive power or the computational complexity of the language. Thus, our main conceptual contribution is a formalization of the semantics of GraphQL. We begin by defining a logical data model that formally captures the notion of a GraphQL graph, as well as the corresponding notion of a GraphQL schema (cf. Section 2). Thereafter, based on our data model, we formalize the semantics of GraphQL queries by using a compositional approach (cf. Section 3) providing a normal form that we heavily use when study the properties of the language.

As our first technical contribution we use our formalization to study the computational complexity of GraphQL (cf. Section 4). We study the classical decision evaluation problem and the enumeration problem, showing that both can be efficiently solved. In particular, we show that the evaluation problem is complete for the class of problems decided in Nondeterministic Logarithmic Space. Moreover, we prove that for queries that satisfy the above mentioned normal form, the enumeration problem can be solved with constant delay. This implies that a server can answer a GraphQL query and send the response byte-by-byte while spending just a constant amount of time between every byte sent.

We also study the problem of computing the size of a GraphQL response (cf. Section 5), showing that it can be solved in polynomial time. That is, even though the size of a query result can be prohibitively large, one can efficiently compute the exact size without executing the query. Our results on polynomial-time size computation plus the constant-delay enumeration provide a robust way of tackling the issues presented by current GraphQL implementations.

In Section 6 we review the related work, briefly comparing GraphQL with other more classical query languages. The conclusions of our work are presented in Section 7. We emphasize that this paper is a substantially extended version of a workshop paper [10] in which we initially presented a fragment of the formalization of the language but no result about normal forms, the complexity of the enumeration or the size computation problems.

2 DATA MODEL

A dataset that is made available via a GraphQL interface can be queried in terms of a so-called *schema* based on which the dataset is represented implicitly as a directed, edge-labeled multigraph with typed nodes and node properties. The nodes in this graph correspond to JSON-style objects that may occur in the query results. The schema associated with a GraphQL interface introduces a notion of types for these objects. Such a type characterizes what fields an object of the given type may have and what values are allowed for each of these fields. The possible values can be restricted to a specific type of scalars or objects. To define the GraphQL query semantics formally we first need to make explicit this logical data model assumed by GraphQL. To this end, this section formalizes the notions of a GraphQL schema and a GraphQL graph. For each concept of the GraphQL specification that our definitions capture, we refer to section of the specification that introduces the concept.

2.1 GraphQL Schema

We consider three infinite countable sets: `Fields` (field names, §2.5 [5]), `Arguments` (argument names, §2.6 [5]), and `Types` (type names, §3.1 [5]). We assume that `Fields`, `Arguments` and `Types` are disjoint and that there exists a finite set `Scalars` (scalar type

```
type Starship {
  id: ID
  name: String
  length: Float
}

interface Character {
  id: ID
  name: String
  friends: [Character]
}

type Droid implements Character {
  id: ID
  name: String
  friends: [Character]
  primaryFunction: String
}

type Human implements Character {
  id: ID
  name: String
  friends: [Character]
  starships: [Starship]
}

enum Episode { NEWHOPE EMPIRE JEDI }

union SearchResult = Human | Droid |
                     Starship
type Query {
  hero(episode: Episode): Character
  search(text: String): [SearchResult]
}

schema {
  query: Query
}
```

Figure 2: Example GraphQL schema in its original syntax

names, §3.1.1 [5]) which is a subset of Types. We also consider a set Vals of scalar values, and a function *values* : $\mathtt{Scalars} \to 2^{\mathtt{Vals}}$ that assigns a set of values to every scalar type.

GraphQL schemas and graphs are defined over finite subsets of the above sets. We assume three finite sets $\mathtt{F} \subset \mathtt{Fields}$, $\mathtt{A} \subset \mathtt{Arguments}$, and $\mathtt{T} \subset \mathtt{Types}$, where T is the disjoint union of $\mathtt{O_T}$ (object types, §3.1.2 [5]), $\mathtt{I_T}$ (interface types, §3.1.3 [5]), $\mathtt{U_T}$ (union types, §3.1.4 [5]) and Scalars, and we denote by $\mathtt{L_T}$ the set $\{[\ \mathtt{t}\ ] \mid \mathtt{t} \in \mathtt{T}\}$ of list types constructed from T (cf. §3.1.7 [5]). We now have everything necessary to define a GraphQL schema over $(\mathtt{F}, \mathtt{A}, \mathtt{T})$.

Definition 2.1. A GraphQL schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ is composed of the following five assignments:

- $\mathit{fields}_{\mathcal{S}} : (\mathtt{O_T} \cup \mathtt{I_T}) \to 2^{\mathtt{F}}$ that assigns a set of fields to every object type and every interface type,
- $\mathit{args}_{\mathcal{S}} : \mathtt{F} \to 2^{\mathtt{A}}$ that assigns a set of arguments to every field,
- $\mathit{type}_{\mathcal{S}} : \mathtt{F} \cup \mathtt{A} \to \mathtt{T} \cup \mathtt{L_T}$ that assigns a type or a list type to every field and argument, where arguments are assigned scalar types; i.e., $\mathit{type}_{\mathcal{S}}(\mathtt{a}) \in \mathtt{Scalars}$ for all $\mathtt{a} \in \mathtt{A}$,
- $\mathit{union}_{\mathcal{S}} : \mathtt{U_T} \to 2^{\mathtt{O_T}}$ that assigns a nonempty set of object types to every union type,
- $\mathit{implementation}_{\mathcal{S}} : \mathtt{I_T} \to 2^{\mathtt{O_T}}$ that assigns a set of object types to every interface.

Additionally, $\mathcal{S}$ contains a distinguished type $\mathit{root}_{\mathcal{S}} \in \mathtt{O_T}$ called the (query) *root type.*

To avoid an overly complex formalization, our definition of a GraphQL schema does not capture the additional notions of *input types* (cf. §3.1.6 [5]) and *non-null types* (cf. §3.1.8 [5]). Moreover, since we are mostly interested in queries, we do not consider *mutation types* (§3.3 [5]). A GraphQL schema is *consistent* if every object type that implements an interface type i defines at least all the fields that i defines. Formally, $\mathcal{S}$ is consistent if $\mathit{fields}_{\mathcal{S}}(\mathtt{i}) \subseteq \mathit{fields}_{\mathcal{S}}(\mathtt{t})$ for every $\mathtt{t} \in \mathit{implementation}_{\mathcal{S}}(\mathtt{i})$. We assume that all GraphQL schemas in this paper are consistent.

Example 2.2. Figure 2 illustrates a GraphQL schema for data about Star Wars movies in the original syntax [5]. This a simplified version of the schema used in one of the official learning resources for GraphQL (see http://graphql.org/learn/schema/). In terms of our formalization, we have a schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ with

$$\mathtt{F} = \{\mathtt{id}, \mathtt{name}, \mathtt{length}, \mathtt{friends}, \mathtt{primaryFunction}, \mathtt{starships}, \mathtt{hero}, \mathtt{search}\},$$

$$\mathtt{A} = \{\mathtt{episode}, \mathtt{text}\}, \quad \text{and} \quad \mathtt{T} = \mathtt{O_T} \cup \mathtt{I_T} \cup \mathtt{U_T} \cup \mathtt{Scalars} \quad \text{such that}$$

$$\mathtt{I_T} = \{\mathtt{Character}\}, \qquad \mathtt{U_T} = \{\mathtt{SearchResult}\},$$
$$\mathtt{O_T} = \{\mathtt{Starship}, \mathtt{Droid}, \mathtt{Human}, \mathtt{Query}\},$$
$$\mathtt{Scalars} = \{\mathtt{ID}, \mathtt{String}, \mathtt{Float}, \mathtt{Episode}\}.$$

As we can see in Figure 2, in the original syntax, object types are defined using the keyword type, and interface and union types with keywords interface and union, respectively. The values for the scalar types are implicit in their names (String, Float) except for ID which is a special type used for unique identifiers (cf. §3.1.1.5 [5]), and Episodes which is an *enum type* such that $\mathit{values}(\mathtt{Episodes}) = \{\mathtt{NEWHOPE}, \mathtt{EMPIRE}, \mathtt{JEDI}\}$. Regarding the functions that compose $\mathcal{S}$ we have that $\mathit{fields}_{\mathcal{S}}$ defines the assignments:

$$\begin{aligned}
\mathtt{Starship} &\to \{\mathtt{id}, \mathtt{name}, \mathtt{length}\},\\
\mathtt{Character} &\to \{\mathtt{id}, \mathtt{name}, \mathtt{friends}\},\\
\mathtt{Droid} &\to \{\mathtt{id}, \mathtt{name}, \mathtt{friends}, \mathtt{primaryFunction}\},\\
\mathtt{Human} &\to \{\mathtt{id}, \mathtt{name}, \mathtt{friends}, \mathtt{starships}\},\\
\mathtt{Query} &\to \{\mathtt{hero}, \mathtt{search}\},
\end{aligned}$$

function $\mathit{args}_{\mathcal{S}}$ defines the assignments:

$$\mathtt{hero} \to \{\mathtt{episode}\}, \qquad \mathtt{search} \to \{\mathtt{text}\},$$

and $\mathit{type}_{\mathcal{S}}$ defines the assignments:

$$\begin{array}{rcl rcl}
\mathtt{id} & \to & \mathtt{ID}, & \mathtt{friends} & \to & [\mathtt{Character}],\\
\mathtt{name} & \to & \mathtt{String}, & \mathtt{starships} & \to & [\mathtt{Starship}],\\
\mathtt{length} & \to & \mathtt{Float}, & \mathtt{primaryFunction} & \to & \mathtt{String}\\
\mathtt{episode} & \to & \mathtt{Episode}, & \mathtt{hero} & \to & \mathtt{Character},\\
\mathtt{text} & \to & \mathtt{String}, & \mathtt{search} & \to & [\mathtt{SearchResult}].
\end{array}$$

The interface and union types are given as follows:

$$\mathit{implementation}_{\mathcal{S}}(\mathtt{Character}) = \{\mathtt{Human}, \mathtt{Droid}\},$$
$$\mathit{union}_{\mathcal{S}}(\mathtt{SearchResult}) = \{\mathtt{Human}, \mathtt{Droid}, \mathtt{Starship}\}.$$

Finally the root type is defined as $\mathit{root}_{\mathcal{S}} = \mathtt{Query}$. In Figure 2 this is defined as the type of the query field under a special schema type.

2.2 GraphQL Graphs

We now define the notion of a GraphQL graph by using the aforementioned domain $(\mathtt{F}, \mathtt{A}, \mathtt{T})$. Informally, a GraphQL graph is a directed, edge-labeled multigraph. Each node in the graph is associated with an object type from $\mathtt{O_T}$ and a set of properties (key-value pairs). The key names of these properties, as well as the edge labels, consist of a field name from F and a (possibly empty) set of arguments, where such an argument is a pair consisting of an argument name from A and a corresponding value. The value of each node property is either a single scalar value or a sequence of scalars. Formally, we define the notion of a GraphQL graph as follows.

Definition 2.3. A *GraphQL graph*, or simply *graph*, over $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ is a tuple $G = (N, E, \tau, \lambda, r)$ with the following elements:

- N is a set of nodes,
- E is a set of edges of the form $(u, \mathtt{f}[\alpha], v)$ where $u, v \in N$, $\mathtt{f} \in \mathtt{F}$, and α is a partial mapping from A to Vals,
- $\tau : N \to \mathtt{O_T}$ is a function that assigns a type to every node,
- λ is a partial function that assigns a scalar value $\mathtt{v} \in \mathtt{Vals}$ or a sequence $[\mathtt{v}_1 \cdots \mathtt{v}_n]$ of scalar values ($\mathtt{v}_i \in \mathtt{Vals}$) to some pairs of the form $(u, \mathtt{f}[\alpha])$ where $u \in N$, $\mathtt{f} \in \mathtt{F}$, and α is a partial mapping from A to Vals,
- $r \in N$ is a distinguished node called the *root* node.

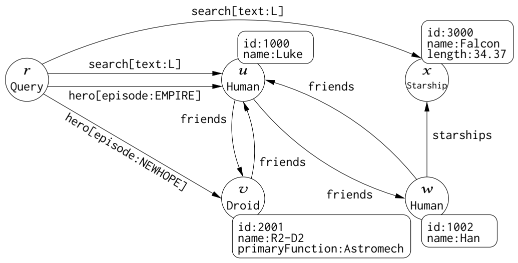


Figure 3: Example GraphQL graph

Example 2.4. Figure 3 illustrates a graph $G = (N, E, \tau, \lambda, r)$ over the domain $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ as given in Example 2.2. G is a simplified version of the graph used in one of the official learning resources for GraphQL (see http://graphql.org/learn/schema/). In this case we have $N = \{r, u, v, w, x\}$ and E contains several edges, including edges $(r, \mathtt{hero[epsiode : EMPIRE]}, u)$ and $(u, \mathtt{friends}, w)$. The type assignment τ is depicted inside every node. For instance $\tau(v) = \mathtt{Droid}$. Function λ is shown as a box beside every node. For instance $\lambda(w, \mathtt{name}) = \mathtt{Han}$, and $\lambda(x, \mathtt{length}) = \mathtt{34.37}$.

Observe that Definition 2.3 introduces the notion of a GraphQL graph independent of any particular GraphQL schema. However, for the purpose of defining queries over such a graph, the graph is assumed to conform to a given schema. Informally, the conditions of conformance to a schema $\mathcal{S}$ imposes on a graph G are summarized as follows: For every edge $(u, \mathtt{f}[\alpha], v)$, field $\mathtt{f}$ is among the field names for the type of u ($\mathtt{f} \in \mathit{fields}_{\mathcal{S}}(\tau(u))$). The type that $\mathcal{S}$ associates with $\mathtt{f}$ must match the type of v ($\mathit{type}_{\mathcal{S}}(\mathtt{f}) = \tau(v)$, or $\tau(v) \in \mathit{implementation}_{\mathcal{S}}(\mathit{type}_{\mathcal{S}}(\mathtt{f}))$, or $\tau(v) \in \mathit{union}_{\mathcal{S}}(\mathit{type}_{\mathcal{S}}(\mathtt{f}))$), and if this type is not a list type, then v is the only node connected to u by an edge with label $\mathtt{f}[\alpha]$. Moreover, for every argument map $\mathtt{a} : \mathtt{v}$ in α, argument name $\mathtt{a}$ must be among the arguments that $\mathcal{S}$ associates with $\mathtt{f}$ ($\mathtt{a} \in \mathit{args}_{\mathcal{S}}(\mathtt{f})$), and value $\mathtt{v}$ must be of the type associated with $\mathtt{a}$ ($\mathtt{v} \in \mathit{values}(\mathit{type}_{\mathcal{S}}(\mathtt{a}))$. In addition to these conditions for the edges in E, there exist similar conditions for the node properties defined by function λ. Finally, the type of the root node should be the root type of $\mathcal{S}$ ($\tau(r) = \mathit{root}_{\mathcal{S}}$). With these intuitions, one can see that the graph in Example 2.4 conforms to the schema described in Example 2.2. Providing a detailed definition of these conditions is straightforward. Due to space limitations, we omit the definition in this paper. Finally, the size of graph G, denoted by $|G|$, is the total number of edges and node properties in G.

We emphasize that our notion of a GraphQL graph is mostly a logical construct needed to base our work on a well-defined foundation. In practice, GraphQL interfaces typically provide access to an underlying database which may be stored using relational technology or in a NoSQL system. The actual data exposed via such an interface can be conceived of as graph-based view of the underlying database. GraphQL graphs are an abstraction of such views that allows us to formalize and study the GraphQL language independent of the technologies used to implement GraphQL interfaces.

3 GRAPHQL LANGUAGE

In this section we provide a formal definition of the GraphQL query language, defining its syntax and semantics over the data model that we introduced in the previous section. Before going into the formal definitions, we give some intuition of the expressions based on which GraphQL queries may be constructed and how these expressions are evaluated. The most basic construction are expressions of the form $\mathtt{f}[\alpha]$ (cf. §2.5 [5]). Informally, when evaluated over a graph, such an expression can be used to match node properties whose name has the same form. Then, assuming the value of the property is a scalar value $\mathtt{v}$, the result of the evaluation is a string of the form $\mathtt{f}:\mathtt{v}$. An alternative to the construction $\mathtt{f}[\alpha]$ is $\ell:\mathtt{f}[\alpha]$ which captures the notion of *"field aliases"* (cf. §2.7 [5]). Such aliases can be used to rename the field names that appear in the query result. To match edges, expressions of the form $\mathtt{f}[\alpha]\{\varphi\}$ can be used, where φ is a subquery to be evaluated in the context of the target nodes. Then, for the case of a single matching edge, the result is a string of the form $\mathtt{f}:\{\rho\}$ with ρ being the string resulting from the evaluation of the subquery φ. On the other hand, if the number of matching edges may be greater than one (which may be the case if the type associated with field $\mathtt{f}$ is a list type), then the result string is of the form $\mathtt{f}:[\{\rho_1\} \cdots \{\rho_n\}]$. Expressions of the form $\mathtt{f}[\alpha]\{\varphi\}$ can also be prefixed with a field alias: $\ell:\mathtt{f}[\alpha]\{\varphi\}$.

```
hero[episode:EMPIRE] {
    name
    friends {
        on Human {
            humanFriend:name
            starships {
                starship:name
                length
            }
        }
        on Droid {
            droidFriend:name
            primaryFunction
        }
    }
}
```

```
hero:{
    name:Luke
    friends:[{
        humanFriend:Han
        starships:[{
            starship:Falcon
            length:34.37
        }]
    }
    {
        droidFriend:R2-D2
        primaryFunction:Astromech
    }]
}
```

Figure 4: GraphQL query (left) and response object (right)

Our query syntax introduces two more constructions: $\mathtt{on}\ \mathtt{t}\{\varphi\}$ and $\varphi_1 \cdots \varphi_n$. While the latter is simply an enumeration of multiple subexpressions whose results are meant to be concatenated, the former captures the notion of a *"type condition"* that is given by what the GraphQL specification refers to as an *"inline fragment"* (cf. §2.8.2 [5]). Hence, $\mathtt{t}$ is either an object type, an interface type, or a union type, and φ is a subquery to be evaluated only for nodes whose associated type is compatible with $\mathtt{t}$.

Readers who are familiar with the query syntax introduced in the GraphQL specification may notice that we do not capture a number of additional language features, namely, (non-inline) *"fragments"* (§2.8 [5]), *"variables"* (§2.10 [5]), and *"directives"* (§2.12 [5]). We emphasize that these features are merely syntactic sugar that a query parser may resolve by using the features captured in the presented syntax. The following definition formalizes our syntax of GraphQL queries.

Definition 3.1. A *GraphQL query*, or simply *query*, over $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ is an expression φ constructed from the following grammar where [,], {, }, :, and on are terminal symbols, $\mathtt{t} \in \mathtt{O}_\mathtt{T} \cup \mathtt{I}_\mathtt{T} \cup \mathtt{U}_\mathtt{T}$, $\mathtt{f} \in \mathtt{F}$, $\ell \in \mathtt{Fields}$, and α represents a partial mapping from $\mathtt{A}$ to $\mathtt{Vals}$.

$$\begin{array}{rccccccc} \varphi & ::= & \mathtt{f}[\alpha] & | & \ell:\mathtt{f}[\alpha] & | & \mathtt{on}\ \mathtt{t}\{\varphi\} & | \\ & & \mathtt{f}[\alpha]\{\varphi\} & | & \ell:\mathtt{f}[\alpha]\{\varphi\} & | & \varphi \cdots \varphi & \end{array}$$

For the sake of conciseness (and in correspondence with the original GraphQL syntax), for sub-expressions of the form $\mathtt{f}[\alpha]$ with α the empty mapping, we just write $\mathtt{f}$. Figure 4 (left) shows an example GraphQL query over the domain $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ in Example 2.2.

In the GraphQL specification [5], queries begin with the optional keyword query followed by an expression as the one introduced in Definition 3.1. That is, the query in Figure 4 (left) is written as `query{hero[episode:EMPIRE]{ ··· }}` (see also Fig. 1). We dropped the `query` keyword to have a simpler recursive syntax.

A notion that we shall use heavily is the size of a query φ, denoted by $|\varphi|$, that we define as the number of field selections and types occurring in φ. This can be formally defined by recursion as follows:

- $|\mathtt{f[}\alpha\mathtt{]}| = |\ell\mathtt{:f[}\alpha\mathtt{]}| = 1$
- $|\mathtt{on\ t\{}\varphi\mathtt{\}}| = |\mathtt{f[}\alpha\mathtt{]\{}\varphi\mathtt{\}}| = |\ell\mathtt{:f[}\alpha\mathtt{]\{}\varphi\mathtt{\}}| = 1 + |\varphi|$
- $|\varphi_1\varphi_2\cdots\varphi_k| = |\varphi_1| + |\varphi_2| + \cdots + |\varphi_k|$

For instance, the query in Figure 4 (left) has size 11.

As for the case of GraphQL graphs, there is a notion of whether a query conforms to a schema $\mathcal{S}$. For instance, if a query begins with an expression of the form $\mathtt{f\{g\{}\varphi\mathtt{\}\}}$, then $\mathtt{f}$ must be a field of the root type in $\mathcal{S}$ ($\mathtt{f} \in \mathit{fields}_{\mathcal{S}}(\mathit{root}_{\mathcal{S}})$), $\mathtt{g}$ must be a field of the type assigned to $\mathtt{f}$ ($\mathtt{g} \in \mathit{fields}_{\mathcal{S}}(\mathit{type}_{\mathcal{S}}(\mathtt{f}))$), and so on. Due to space limitations we do not include all the formal requirements here, but they are as straightforward to define as for the case of queries.

As a last preliminary for formalizing the semantics of GraphQL queries we require a definition of the notion of a result that a GraphQL query may return.

Definition 3.2. A *GraphQL response object* is an expression ρ constructed from the following grammar where {, }, [,], :, and `null` are terminal symbols, ε denotes the empty word, $\ell \in \mathtt{Fields}$, and $\mathtt{v}, \mathtt{v}_1, \ldots, \mathtt{v}_n \in \mathtt{Vals}$:

$$\begin{array}{llllllllll} \rho & ::= & \ell\mathtt{:v} & \mid & \ell\mathtt{:[v}_1\cdots\mathtt{v}_n\mathtt{]} & \mid & \ell\mathtt{:null} & \mid \\ & & \ell\mathtt{:\{}\rho\mathtt{\}} & \mid & \ell\mathtt{:[\{}\rho\mathtt{\}}\cdots\mathtt{\{}\rho\mathtt{\}]} & \mid & \rho\cdots\rho & \mid & \varepsilon \end{array}$$

Figure 4 (right) shows an example response object. Such objects are strings over alphabet $\Sigma = \mathtt{Fields} \cup \mathtt{Vals} \cup \{\{,\},[,],:,\mathtt{null}\}$, and thus we can define the size of a response object ρ, denoted by $|\rho|$, simply as the number of symbols of Σ occurring in ρ. For instance, the size of the response object in Figure 4 is 36. Similarly as for the case of queries, response objects in the official specification begin with the keyword `data` (see Fig. 1(b)). We have also dropped that keyword to have a simpler syntax.

As a final note on the syntax of queries and response objects, notice that both have a *tree structure*. Thus, one can intuitively talk, for example, about *root* or *leaf* fields of a query or a response object, or even about *children* of a field in a query.

Field collection. Before going into the semantics we need the additional notion of *collecting fields* (cf. §6.3.2 [5]). The main idea is that repeated fields in a query/response should not be considered twice. For instance, a GraphQL query will never result in a response object of the following form:

`droid:{ name:C3PO } ship:{ length:30.0 } droid:{ pF:Protocol }.`

Instead, if this is the data to be returned, the response object will be:

`droid:{ name:C3PO pF:Protocol} ship:{ length:30.0 }.`

Notice how the two occurrences of the field `droid` are merged collecting the subfields `name` and `pF` into a single group while maintaining their relative order. The example should clarify why this process is called field collection in the GraphQL specification. To formalize this we use a recursive function collect(·) over response objects. For the sake of space we do not include the details of this function, but it can be easily implemented as a recursive procedure over a response object following the intuition in the example above.

We are now ready to introduce the semantics of GraphQL queries. In what follows we always assume a fixed given schema $\mathcal{S}$.

Definition 3.3. Let $G = (N, E, \tau, \lambda, r)$ be a graph and φ a query, both conforming to a schema $\mathcal{S}$ over $(\mathtt{F}, \mathtt{A}, \mathtt{T})$. The *evaluation of φ over G from node $u \in N$*, denoted by $[\![\varphi]\!]^u_G$, is a GraphQL response object that is defined recursively as shown in Figure 5. The *evaluation of φ over G*, denoted by $[\![\varphi]\!]_G$, is simply $[\![\varphi]\!]^r_G$.

Example 3.4. For the GraphQL graph G in Example 2.4, and the query φ and response object ρ in Figure 4, we have that $\rho = [\![\varphi]\!]_G$.

Equivalences and normal forms

Let φ_1 and φ_2 be queries that conform to a schema $\mathcal{S}$. We say that φ_1 and φ_2 are equivalent, denoted by $\varphi_1 \equiv \varphi_2$, if for every graph G that conforms to $\mathcal{S}$ it holds that $[\![\varphi_1]\!]_G = [\![\varphi_2]\!]_G$. We need two additional notions that shall be useful for algorithms and complexity analysis.

Definition 3.5. A GraphQL query φ is in *ground-typed normal form* if it satisfies the following grammar, where $\mathtt{t} \in \mathtt{O_T}$.

$$\begin{array}{lll} \varphi & ::= & \psi\cdots\psi \;\mid\; \chi\cdots\chi \\ \psi & ::= & \mathtt{on\ t\ \{}\ \chi\cdots\chi\ \mathtt{\}} \\ \chi & ::= & \mathtt{f[}\alpha\mathtt{]} \;\mid\; \ell\mathtt{:f[}\alpha\mathtt{]} \;\mid\; \mathtt{f[}\alpha\mathtt{]\{}\ \varphi\ \mathtt{\}} \;\mid\; \ell\mathtt{:f[}\alpha\mathtt{]\{}\ \varphi\ \mathtt{\}} \end{array}$$

That is, intuitively, φ is in *ground-typed normal form* if the following three conditions are satisfied: (1) for every expression of the form `on t { ... }` that occurs in φ, it holds that $\mathtt{t} \in \mathtt{O_T}$, (2) expressions of the form `on t { ... }` do not occur mixed with regular field selections, and (3) an expression `on t′ { ... }` does not occur immediately inside another `on t { ... }` expression. Finally, we introduce a notion that focuses on possible redundancy in GraphQL queries.

Definition 3.6. A GraphQL query φ is *non-redundant* if it satisfies the following condition. For every subexpression of φ of the form $\varphi_1\cdots\varphi_k$ there are no indexes $i, j \in \{1, \ldots, k\}$ such that $i \neq j$ and

- $\varphi_i = \varphi_j = \mathtt{f[}\alpha\mathtt{]}$, or
- $\varphi_i = \varphi_j = \ell\mathtt{:f[}\alpha\mathtt{]}$, or
- $\varphi_i = \mathtt{f[}\alpha\mathtt{]\{}\beta\mathtt{\}}$ and $\varphi_j = \mathtt{f[}\alpha\mathtt{]\{}\gamma\mathtt{\}}$, or
- $\varphi_i = \ell\mathtt{:f[}\alpha\mathtt{]\{}\beta\mathtt{\}}$ and $\varphi_j = \ell\mathtt{:f[}\alpha\mathtt{]\{}\gamma\mathtt{\}}$, or
- $\varphi_i = \mathtt{on\ t\ \{}\beta\mathtt{\}}$ and $\varphi_j = \mathtt{on\ t\ \{}\gamma\mathtt{\}}$.

Example 3.7. The query in Figure 4 is a non-redundant query in ground-typed normal form.

Theorem 3.8. *For every query φ that conforms to a schema $\mathcal{S}$ there exists a non-redundant query φ' in ground-typed normal form such that $\varphi \equiv \varphi'$.*

Theorem 3.8 can be obtained by rewriting queries using equivalence rules. Due to space constraints, we cannot include these rules in this paper; we will provide them in the full version of the paper.

In the rest of the paper we assume that every GraphQL query is a non-redundant query in ground-typed normal form. One of the main properties of such queries is that they produce a unique response object without the need of the collect(·) operator. More formally, let $\langle\!\langle\varphi\rangle\!\rangle_G$ be an evaluation function for queries defined in exactly the same way as $[\![\varphi]\!]_G$ in Definition 3.3 but replacing the last rule in Figure 5 by $\langle\!\langle\varphi_1\cdots\varphi_k\rangle\!\rangle^u_G = \langle\!\langle\varphi_1\rangle\!\rangle^u_G\cdots\langle\!\langle\varphi_k\rangle\!\rangle^u_G$, that is, without using collect(·). It is not difficult to prove that if φ is a non-redundant query in ground-typed normal form, then $[\![\varphi]\!]_G = \langle\!\langle\varphi\rangle\!\rangle_G$ for every graph G. We shall exploit this property in the next sections.

$$[\![\mathtt{f}[\alpha]]\!]^u_G = \begin{cases} \mathtt{f}\!:\!\lambda(u,\mathtt{f}[\alpha]) & \text{if } (u,\mathtt{f}[\alpha]) \in \mathrm{dom}(\lambda) \\ \mathtt{f}\!:\!\mathtt{null} & \text{else.} \end{cases}$$

$$[\![\ell\!:\!\mathtt{f}[\alpha]]\!]^u_G = \begin{cases} \ell\!:\!\lambda(u,\mathtt{f}[\alpha]) & \text{if } (u,\mathtt{f}[\alpha]) \in \mathrm{dom}(\lambda) \\ \ell\!:\!\mathtt{null} & \text{else.} \end{cases}$$

$$[\![\mathtt{f}[\alpha]\{\varphi\}]\!]^u_G = \begin{cases} \mathtt{f}\!:\![\{[\![\varphi]\!]^{v_1}_G\}\cdots\{[\![\varphi]\!]^{v_k}_G\}] & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \in \mathsf{L}_\mathsf{T} \text{ and } \{v_1,\dots,v_k\} = \{v_i \mid (u,\mathtt{f}[\alpha],v_i) \in E\} \\ \mathtt{f}\!:\!\{\, [\![\varphi]\!]^{v}_G \,\} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathsf{L}_\mathsf{T} \text{ and } (u,\mathtt{f}[\alpha],v) \in E \\ \mathtt{f}\!:\!\mathtt{null} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathsf{L}_\mathsf{T} \text{ and there is no } v \in N \text{ s.t. } (u,\mathtt{f}[\alpha],v) \in E \end{cases}$$

$$[\![\ell\!:\!\mathtt{f}[\alpha]\{\varphi\}]\!]^u_G = \begin{cases} \ell\!:\![\{[\![\varphi]\!]^{v_1}_G\}\cdots\{[\![\varphi]\!]^{v_k}_G\}] & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \in \mathsf{L}_\mathsf{T} \text{ and } \{v_1,\dots,v_k\} = \{v_i \mid (u,\mathtt{f}[\alpha],v_i) \in E\} \\ \ell\!:\!\{\, [\![\varphi]\!]^{v}_G \,\} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathsf{L}_\mathsf{T} \text{ and } (u,\mathtt{f}[\alpha],v) \in E \\ \ell\!:\!\mathtt{null} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathsf{L}_\mathsf{T} \text{ and there is no } v \in N \text{ s.t. } (u,\mathtt{f}[\alpha],v) \in E \end{cases}$$

$$[\![\mathtt{on\ t}\{\varphi\}]\!]^u_G = \begin{cases} [\![\varphi]\!]^u_G & \text{if } \mathtt{t} \in \mathsf{O}_\mathsf{T} \text{ and } \tau(u) = \mathtt{t}, \text{ or } \mathtt{t} \in \mathsf{I}_\mathsf{T} \text{ and } \tau(u) \in \mathit{implementation}_{\mathcal{S}}(\mathtt{t}), \text{ or} \\ & \mathtt{t} \in \mathsf{U}_\mathsf{T} \text{ and } \tau(u) \in \mathit{union}_{\mathcal{S}}(\mathtt{t}) \\ \varepsilon & \text{in other case.} \end{cases}$$

$$[\![\varphi_1 \cdots \varphi_k]\!]^u_G = \mathrm{collect}([\![\varphi_1]\!]^u_G \cdots [\![\varphi_k]\!]^u_G)$$

Figure 5: Semantics of a GraphQL query.

4 THE COMPLEXITY OF GRAPHQL

In this section we study the complexity of two classical decision problems in the context of GraphQL, namely, the evaluation problem and the enumeration problem, showing that both can be solved efficiently. For this analysis we make the following assumption: Let G be a GraphQL graph, u be a node, and $\mathtt{f}[\alpha]$ be an edge label. We assume that one can access the list of $\mathtt{f}[\alpha]$-neighbors of u in time $O(1)$, and one can access the $\mathtt{f}[\alpha]$-property of a node in time $O(1)$. Although this is a standard assumption for graph databases in a RAM computational model, we stress that a GraphQL graph is usually implemented as a *view* over another data source and, thus, the time required to access neighbors and data may depend on the underlying data storage. Our assumption allows us to study the two decision problems independent of implementation-specific peculiarities.

Classical query languages, such as SQL or Relational Algebra, take as inputs a query and a database and produce a set of tuples as output. For these languages the standard way of defining a decision problem is the following: given a query Q, a database D, and a candidate tuple t, check if t is part of the evaluation of Q over D [21]. In contrast to classical languages, the result of a GraphQL query is not a set of tuples but a single response object. To define a similar decision problem for GraphQL, we consider the data values occurring in response objects. For example, in the object

```
droid:{ name:C3PO pF:Protocol} ship:{ length:30.0 }
```

the values that occur are `C3PO`, `Protocol` and `30.0`. Formally, we define the following decision problem.

Problem:	GraphQL-Eval
Input:	GraphQL query φ, graph G, and value $\mathsf{v} \in \mathtt{Vals}$
Ouput:	Does v occur in $[\![\varphi]\!]_G$?

We next show that GraphQL-Eval is complete for the class of problems that can be decided in nondeterministic logarithmic space.

Theorem 4.1. GraphQL-Eval *is NL-complete.*

Proof. (Sketch) The proof of the membership in NL is based on characterizing the evaluation process as several reachability tests. Recall first that a GraphQL query can be seen as a tree. Moreover, one can traverse a query by following its tree structure, that is, going from one label up to its parent, down to one of its children, or left/right to its siblings, with a logspace machine by using standard techniques (see e.g. [8] Section 2.5). Thus, to show membership in NL we first guess a position, say p, in φ corresponding to an expression of the form $\mathtt{f}[\alpha]$ or $\ell\!:\!\mathtt{f}[\alpha]$. We also guess a node, say u, in G. Notice that to store p and u we only need logarithmic space. The intuition is that p represents the field in φ that when evaluated from u produces the value v. This last property holds if and only if either $\lambda(u,\mathtt{f}[\alpha]) = \mathsf{v}$ or $\lambda(u,\mathtt{f}[\alpha])$ is a list containing v. Both options can be checked by simply inspecting G. To complete the proof we consider the path P (sequence of field names and type restrictions) in the tree representation of φ that leads to position p. The last step in the proof is to check that node u can be reached from the root node in G by following path P which can be done in NL by a standard reachability test.

NL-hardness follows from the reachability problem in directed graphs. Given a directed graph G with k nodes and two nodes u and v, we create a GraphQL graph G' from G by setting u as the root node, and adding a field label $\mathtt{e}$ to every edge. Moreover, for every node in G' we add a property $\mathtt{a}$ with value $\mathtt{1}$ except for node v in which $\mathtt{a}$ is associated with value $\mathtt{2}$. Then, we consider the sequence of queries constructed recursively as $\alpha_1 = \mathtt{a}$ and $\alpha_i = \mathtt{a\ e}\{\alpha_{i-1}\}$, and the query $\varphi = \alpha_k$. That is, φ is the query $\mathtt{a\ e\{a\ e\{a\ e\{\ \cdots\ \}\}\}}$, where $\mathtt{a}$ is repeated k times and $\mathtt{e}$ repeated $k-1$ times. It is not difficult to argue that G' and φ can be constructed from G by using only logarithmic space. Moreover, value $\mathtt{2}$ occurs in $[\![\varphi]\!]_{G'}$ if and only if v is reachable from u in G. □

Although it is theoretically interesting to pinpoint the exact complexity of the GraphQL evaluation problem, the previous result does not give a specific hint on how the whole evaluation of a query can be actually computed in practice. We next prove that for non-redundant queries in ground-typed normal form, the complete evaluation can be done in time linear with respect to the size of the output. Actually, in proving this result we show something even stronger: the complete evaluation of a GraphQL query can be constructed symbol-by-symbol with only constant-time delay between each symbol.

Theorem 4.2. *Let G be a GraphQL graph and φ a non-redundant query in ground-typed normal form. Then, $\rho = [\![\varphi]\!]_G$ can be computed such that ρ is produced symbol-by-symbol with constant-time delay between each symbol.*

Proof. (Sketch) We consider a fixed GraphQL schema $\mathcal{S}$. Let φ be a non-redundant query in ground-typed normal form, and G be a graph, both conforming to $\mathcal{S}$. We describe a recursive algorithm Enumerate that receives a subquery of φ and a node u. The initial call is Enumerate(φ, r) where r is the root node. For the recursive case we assume that φ follows the grammar in Definition 3.5. Thus, consider a call Enumerate(χ, u) with χ a subquery of φ given by the third rule in Definition 3.5. Assume first that χ is of the form $\mathtt{f}[\alpha]$. If there is a value $\mathsf{V} = \lambda(u, \mathtt{f}[\alpha])$ in G, then Enumerate outputs $\mathtt{f}\!:\!\mathsf{V}$, otherwise it outputs $\mathtt{f}\!:\!\mathtt{null}$, and returns. Assume now that χ is of the form $\mathtt{f}[\alpha]\{\varphi'\}$. Enumerate first outputs expression $\mathtt{f}\!:$ and then it proceeds depending on the following cases:

(1) If $type_{\mathcal{S}}(\mathtt{f}) \in \mathsf{L}_\mathsf{T}$, then Enumerate outputs the symbol `[`. Next, for every v such that $(u, \mathtt{f}[\alpha], v) \in E$, it first outputs the symbol `{`, then calls Enumerate(φ', v), and then outputs the symbol `}`. Finally, it outputs the symbol `]` and returns.
(2) If $type_{\mathcal{S}}(\mathtt{f}) \notin \mathsf{L}_\mathsf{T}$, then we have two cases depending on whether there is a v such that $(u, \mathtt{f}[\alpha], v) \in E$ or not. If there is no such v, then Enumerate simply outputs `null` and returns. Otherwise it first outputs the symbol `{`, then calls Enumerate(φ', v), and then outputs the symbol `}`.

The cases in which χ is of the form $\ell\!:\!\mathtt{f}[\alpha]$ or $\ell\!:\!\mathtt{f}[\alpha]\{\varphi'\}$ are similar but Enumerate outputs first $\ell\!:$ instead of $\mathtt{f}\!:$. Notice that in all these cases, Enumerate is just following the semantics of GraphQL as defined in Definition 3.3. The really important cases are when we have expressions constructed from the first or second rule in Definition 3.5. Hence, consider a subquery of the form $\chi_1\chi_2 \cdots \chi_k$ where every χ_i is constructed from the third rule in Definition 3.5. Given that φ is non-redundant and in ground-typed normal form, we do not need the collect$(\cdot)$ operator and, thus, Enumerate$(\chi_1\chi_2 \cdots \chi_k, u)$ can simply call Enumerate(χ_i, u) one by one for every $i = 1, 2, \ldots, k$ and produce the desired output. Finally, consider a subquery $\psi_1\psi_2 \cdots \psi_k$ where every ψ_i is of the form $\mathtt{on}\ \mathtt{t}_i\ \{\varphi_i\}$. Given that φ is non-redundant we know that $\mathtt{t}_i \neq \mathtt{t}_j$ for $i \neq j$. This implies that k is a constant value bounded by the number of types mentioned in $\mathcal{S}$. Moreover, since φ is ground typed, we have $\mathtt{t}_i \in \mathsf{O}_\mathsf{T}$ for every i. With these observations the call Enumerate$(\psi_1\psi_2 \cdots \psi_k, u)$ can proceed as follows. Search for an index i such that $\tau(u) = \mathtt{t}_i$. If such an index i exists, then call Enumerate(φ_i, u). If the index does not exist, just return.

To see that there is a constant-time delay between any two symbols produced by Enumerate, first notice that every call, except for the last case considered above, outputs at least one symbol as soon as it is called and at least one symbol just before it returns. Moreover, every recursive call to Enumerate is performed after a constant time upon its parent call. Finally, to consider the last case in the previous paragraph, given that φ is non-redundant and in ground-typed normal form we have that an expression of the form $\mathtt{on}\,\mathtt{t}'\,\{\ \ldots\ \}$ does not occur immediately inside another $\mathtt{on}\,\mathtt{t}\,\{\ \ldots\ \}$ expression. This ensures that there are no two consecutive calls to Enumerate that do not produce any output. □

Computing all the components of the evaluation is usually called the *enumeration problem* [3, 13, 17]. Thus, Theorem 4.2 shows that the enumeration problem for GraphQL can be solved with constant delay. As an immediate corollary we obtain the following.

Corollary 4.3. *Let G be a GraphQL graph and φ a non-redundant query in ground-typed normal form. Then, $\rho = [\![\varphi]\!]_G$ can be computed in time linear with respect to $|\rho|$.*

5 THE SIZE OF A GRAPHQL RESPONSE

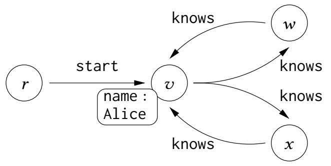
Based on our results in the previous section, we may conclude that one of the main sources of complexity in evaluating GraphQL queries is the size of a query response object. In this section we prove that even for very simple cases this object might be prohibitively large. We begin by stating an exponential upper bound.

Proposition 5.1. *For every GraphQL query φ and GraphQL graph G it holds that $[\![\varphi]\!]_G$ is of size $O(|G|^{|\varphi|})$.*

The upper bound can be proven by a simple induction argument. We next show that this upper bound is tight.

Proposition 5.2. *For every $n \geq 0$, there exists a graph G and a query φ such that G is of size 6 and φ is of size $2(n + 1)$, but the size of $[\![\varphi]\!]_G$ is greater than 2^n.*

Proof. Let G be the following graph with root node r.

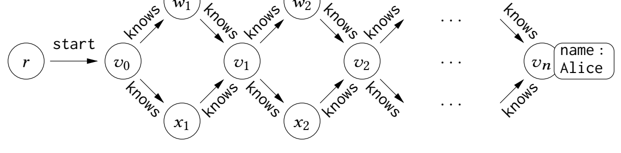


Consider now the queries given by the recurrence $\alpha_0 = \mathtt{name}$, and $\alpha_i = \mathtt{knows}\ \{\ \mathtt{knows}\ \{\ \alpha_{i-1}\ \}\ \}$ for every $i > 0$. Define φ as $\mathtt{start}\,\{\ \alpha_n\ \}$. Then, the size of φ is $2n + 2$ but the evaluation of φ over G is of size exponential in n; more precisely, the name `Alice` occurs 2^n times in $[\![\varphi]\!]_G$. □

One may think that the previous result is produced solely by the presence of directed cycles in the graph. As the next result shows, one can also obtain an exponential blow up even for acyclic graphs.

Proposition 5.3. *For every $n \geq 0$, there exists an acyclic graph G and a query φ such that G is of size $4n + 2$ and φ is of size $2n + 2$, but the size of $[\![\varphi]\!]_G$ is greater than 2^n.*

Proof. Let G be the following graph with root node r.



Furthermore, let φ be as defined in the proof of Proposition 5.2. Then G is of size $4n + 2$, φ is of size $2n + 2$, but $[\![\varphi]\!]_G$ is of size exponential in n; the name `Alice` occurs 2^n times in $[\![\varphi]\!]_G$. □

A natural question at this point is how can we avoid obtaining a response object of exponential size. We prove next that we have at least two options: bound the number of different *walks* in the graph, or bound the *nesting depth* of queries. A walk in a graph is similar to a path but it is allowed to repeat edges. Notice that this implies that a graph with a cycle has an unbounded number of walks. The nesting depth of a query can be defined intuitively as the maximum number of nested curly braces in the query expression. For instance, the query in Figure 4 has nesting depth 4.

Theorem 5.4. *Let G be a graph with root node r, and φ a GraphQL query. Let K be a constant value not depending on the size of G or φ. Consider the following two properties.*

(1) *For every node v in G the number of different walks from r to v is bounded by K.*
(2) *The nesting depth of φ is bounded by K.*

For G and φ satisfying either (1) or (2) we have that the evaluation $[\![\varphi]\!]_G$ is of size $O(|G|^K \cdot |\varphi|)$.

Proof. (Sketch) For case (1), let $\rho = [\![\varphi]\!]_G$. We know that every data value in ρ corresponds to a property of some node in G and, moreover, every path in ρ corresponds to a walk in G from the root node. Thus, the maximum number of data values appearing in ρ is bounded by $|G|^K$. Furthermore, for every such value, the length of its path in ρ is bounded by $|\varphi|$, which gives us the $O(|G|^K \cdot |\varphi|)$ bound. For case (2) the result follows from a simple induction argument. □

Property (2) is exactly one of the restrictions that the GraphQL interface of Github imposes to ensure a reasonable output size (see Sec. 1) [7]; there also exists a software library [20] that can be used to integrate the same type of restriction into any other GraphQL server.

While property (1) or property (2) may be applied as a restriction to avoid exponential blow up of query results, we emphasize that both properties are more restrictive than necessary. That is, there are cases in which the properties are not satisfied but query results can still be of polynomial size only. For instance, we recall our initial Github experiment (cf. Sec. 1) where the owner of some of the Github repositories listed for a Github user with login "danbri" was "danbri" himself. Hence, the data exposed via Github's GraphQL interface contains cycles and, thus, does not satisfy property (1). Now, for the sequence of queries in our experiment (with the level-1 and the level-2 versions illustrated in Figures 1(a) and 1(c), respectively), consider a variation of the queries that uses `first:1` (instead of `first:2` or `first:5`). It is easy to see that the result size of the so-changed queries grows linearly with the level of the queries. Hence, for these queries we may permit an arbitrarily deep nesting without having to expect an exponential result size blow up. In this case, enforcing property (2) is too restrictive.

In addition to restricting the nesting depth of queries, other approaches are used in practice to identify queries whose computation could be too resource intensive. For instance, Github combines the nesting-depth restriction (using a K of 25) with the following restriction. For every subquery of the form $\mathtt{f}[\alpha]\{\varphi\}$ or $\ell{:}\mathtt{f}[\alpha]\{\varphi\}$ for which the type of field `f` is a list type (i.e., $type_{\mathcal{S}}(\mathtt{f}) \in \mathsf{L}_\mathsf{T}$), the arguments α must contain either the argument named `first` or the argument named `last`, with a value that is an integer from 1 to 100. Moreover, based on these argument values, the Github GraphQL interface computes the maximum number of possible result nodes at each level of nesting depth of a given query. Queries for which this number is greater than 500,000 at some level are rejected [7].

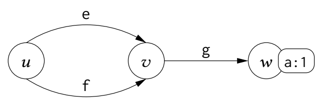
Note that this restriction has not been sufficient to prevent the overly resource intensive attempt to execute the level-6 and the level-7 version of the test queries in our experiment in which we used `first:5` (i.e., Figure 1(e)). On the other hand, the restriction may also be too restrictive in various cases. For instance, there may be a query for which the maximum number of *possible* result nodes at some level of nesting depth is greater than 500,000, but if the query would be executed, the actual number of result nodes may turn out to be significantly smaller and not cause any trouble at all.

We found two more software libraries [1, 14] that aim to estimate some notion of "complexity" or "cost" of GraphQL queries. These estimation approaches take into account different cost factors that a user may associate with the various elements of the schema used. However, these approaches suffer from the same problem as the restrictions imposed by Github's GraphQL interface: They may easily overestimate the cost of a query by a large degree.

Computing the exact size of a GraphQL response. As shown in Theorem 5.4, in order to avoid returning response objects of exponential size, GraphQL service providers must impose severe restrictions on the structure of the queries or the data. Moreover, these restrictions can fail in both directions: discarding settings in which an efficient evaluation is possible and allowing settings in which a complete evaluation is too resource intensive. Fortunately, we can give a more elegant solution by showing that the exact size of the complete evaluation of a GraphQL query can be computed efficiently without the need of evaluating the whole query. We formalize this result in the following theorem.

Theorem 5.5. *Let G be a GraphQL graph and φ a non-redundant query in ground-typed normal form. The size of $[\![\varphi]\!]_G$ can be computed in time $O(|G| \cdot |\varphi|)$.*

To prove the theorem we present a dynamic programming strategy in Algorithms 1 and 2. Let φ be a query and G a graph. The idea of procedure `Label` in Algorithm 2 is to label every node u with all subqueries ψ of φ for which we already know the size of $[\![\psi]\!]^u_G$. In that way we never visit a node twice for the same subquery. We maintain two structures: $\mathtt{labels}[u]$ to store (pointers to) the subqueries, and $\mathtt{size}[u,\psi]$ to store the size of $[\![\psi]\!]^u_G$. To briefly illustrate how the algorithm works, consider the following graph


$$\begin{tikzcd} u \arrow[r, bend left, "\mathsf{e}"] \arrow[r, bend right, "\mathsf{f}"'] & v \arrow[r, "\mathsf{g}"] & w\ \boxed{\mathsf{a:1}} \end{tikzcd}$$

and query φ given by `e { g { a } } f { g { a } }`. Assume that at some point during the execution of `Label` we visit w with subquery `a` for the first time. Then, we store `a` in $\mathtt{labels}[w]$ (line 2 in Algorithm 2) and set $\mathtt{size}[w,\mathtt{a}]$ to 3 (line 4) because the evaluation is `a:1`, which has three symbols. Assume now that we visit v with subquery `g{a}`. We first store this expression in $\mathtt{labels}[v]$ and we recursively call $\mathtt{Label}(G, w, \mathtt{a})$ (line 9) because

Algorithm 1 GraphQLSize(G, φ)

```
Require: non-redundant $\varphi$ in ground-typed normal form
1: for all node $u$ in $G$ do labels$[u] := \emptyset$
2: for all node $u$ in $G$ and $\psi$ sub expression of $\varphi$ do size$[u, \psi] := 0$
3: Label$(G, r, \varphi)$, where $r$ is the root node in $G$
4: return size$[r, \varphi]$
```

Algorithm 2 Label(G, u, φ)

```
1: if $\varphi \notin$ labels$[u]$ then
2:     labels$[u]$ := labels$[u] \cup \{\varphi\}$
3:     if $\varphi$ = f$[\alpha]$ or $\varphi = \ell$: f$[\alpha]$ then
4:         if $(u$, f$[\alpha]) \in \mathrm{dom}(\lambda)$ then size$[u, \varphi] := |\lambda(u$, f$[\alpha])| + 2$
5:         else size$[u, \varphi] := 3$
6:     else if $\varphi$ = f$[\alpha]\{\varphi'\}$ or $\varphi = \ell$: f$[\alpha]\{\varphi'\}$ then
7:         $V := \{v \mid (u$, f$[\alpha], v) \in E\}$
8:         for all $v$ in $V$ do
9:             Label$(G, v, \varphi')$
10:            size$[u, \varphi]$ += size$[v, \varphi']$ + 2
11:        if $type_{\mathcal{S}}$(f) $\in$ L$_{\mathrm{T}}$ then size$[u, \varphi]$ += 4
12:        else if $type_{\mathcal{S}}$(f) $\notin$ L$_{\mathrm{T}}$ and $|V| \neq 0$ then size$[u, \varphi]$ += 2
13:        else size$[u, \varphi] := 3$
14:    else if $\varphi$ = on t $\{\psi\}$ then
15:        if $\tau(u)$ = t then
16:            Label$(G, u, \psi)$
17:            size$[u, \varphi]$ := size$[u, \psi]$
18:    else if $\varphi = \varphi_1 \cdots \varphi_k$ then
19:        for all $i \in \{1, \ldots, k\}$ do
20:            Label$(G, u, \varphi_i)$
21:            size$[u, \varphi]$ += size$[u, \varphi_i]$
```

$(v, \mathrm{g}, w) \in E$. Given that a $\in$ labels$[w]$, the process immediately returns without any additional computation (line 1). Next, we increment size$[v$, g{a}$]$ with size$[w$, a$]+2 = 5$ (line 10), and finally add 2 (line 12) to obtain size$[v$, g{a}$] = 7$, which is exactly the size of the evaluation g:{a:1} = $[\![$g{a}$]\!]^v_G$. With a similar analysis we obtain that the algorithm produces the value size$[u$, e{g{a}}$] = 11$, which is the number of symbols in e:{g:{a:1}}. What is more interesting is the call to Label with arguments u and f{g{a}}. Notice that this call produces a recursive call to Label$(G, v$, g{a}$)$, but since g{a} $\in$ labels$[v]$, we already have the correct size in size$[v$, g{a}$]$ and we do not need to do any additional computation. Finally, the value size$[v$, g{a}$] = 7$ is used to set size$[u$, f{g{a}}$]$ to value 11. All this computation can be used to obtain the size of $[\![\varphi]\!]^u_G$ which is size$[u$, e{g{a}}$]$ + size$[u$, f{g{a}}$] = 22$ (line 21) which is the number of symbols of e:{g:{a:1}} f:{g:{a:1}}. Algorithm 1 initializes all the needed structures and makes the initial call with the whole query and the root node. To see that Algorithm 1 works in time $O(|G| \cdot |\varphi|)$ just notice that the number of subqueries is linear with respect to the query, and every node is visited at most once per subquery. Moreover, when visiting a node u, the number of steps is at most the number of outgoing edges from u.

6 RELATED WORK

A natural question is how our results compare with results for classical query languages. Given the tree-like structure of GraphQL queries, an immediate candidate to compare with is the language of acyclic conjunctive queries (ACQs) [8, 22]. Conjunctive queries (CQs) corresponds to the SELECT-FROM-WHERE fragment of SQL. The further acyclic restriction in ACQs ensures the existence of a *join tree* that allows one to efficiently evaluate the query [22].

In terms of expressiveness, it is not difficult to encode a GraphQL query into an ACQ, although some special care has to be put in the final construction of the GraphQL response object from the tuples of values produced by an ACQ. There is no formal work on proving properties of such an encoding, but practitioners have already started using these types of methods when evaluating GraphQL [19]. Despite the existence of an encoding, the classical ACQs complexity results do not directly entail the results in this paper. In terms of the evaluation problem, Gottlob et al. [8] have shown that for ACQs the problem is complete for LOGCFL, which is the class of problems that can be reduced in logarithmic space to a context-free language [11]. Since it is believed that NL $\subsetneq$ LOGCFL, the membership of GraphQL-Eval in NL shows an important difference in terms of complexity theory. In terms of the enumeration problem, there is a substantial amount of work on restrictions that allow a constant-delay evaluation for ACQs and related languages [2, 4, 12, 17, 18]. As mentioned in [17], "it is very unlikely that constant-delay enumeration can be achieved for all queries in ACQ" and, thus, constant-delay results impose restrictions over the structure of the queries [2, 17] or of the queried data [4, 12, 18]. In contrast, Theorem 4.2 proves the constant-delay result for GraphQL in general, requiring only a specific normal form for queries.

Pichler and Skritek [15] presented a fairly complete picture of the complexity of counting the number of tuples in the evaluation of an ACQ. This problem is closely related to the problem of computing the size of the evaluation of a GraphQL query. In [15] the authors proved that for general ACQs the problem is intractable, and they presented a polynomial-time algorithm for the case of ACQs without existential variables. It is not clear that a GraphQL query can be encoded as a single ACQ without existential variables, but even if possible, applied to our scenario, the algorithm in [15] would work in time $O(|G|^2 \cdot |\varphi|)$ (see Theorem 1 in [15]), while our algorithm works in time linear with respect to $|G|$.

7 CONCLUSIONS

GraphQL is becoming increasingly popular as an alternative to REST-based interfaces which have dominated the Web-API scenario for more than 10 years. In the spirit of classical query languages, GraphQL uses a schema to describe the organization of the data and a declarative query language to allow clients to access this data.

We have embarked on a systematic study of GraphQL providing a full formalization of the semantics of its query language based on a logical data model. Given this formalization, we have also studied the complexity of the language, in particular the evaluation, enumeration, and size-computation problems, showing that all of them can be efficiently solved. While our conceptual contributions can be used to further study the language form a theoretical point of view, our technical contributions can also help developers to implement more robust GraphQL interfaces on the Web.

Acknowledgements. Olaf Hartig's work has been funded by the CENIIT program at Linköping University (project number 17.05). Jorge Pérez is supported by the Millenium Institute for Foundational Research on Data, and ENLACE-Fondecyt VID-UChile.

REFERENCES

[1] 4Catalyzer Corporation. 2017. GraphQL Validation Complexity. https://github.com/4Catalyzer/graphql-validation-complexity/. (2017).

[2] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings.* 208–222.

[3] Nadia Creignou, Markus Kröll, Reinhard Pichler, Sebastian Skritek, and Heribert Vollmer. 2017. On the Complexity of Hard Enumeration Problems. In *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings.* 183–195.

[4] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. 2014. Enumerating answers to first-order queries over databases of low degree. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014.* 121–131.

[5] Facebook, Inc. 2016. GraphQL. Working Draft, Oct. 2016. Online at http://facebook.github.io/graphql, retrieved on Dec. 12, 2016. (Oct. 2016).

[6] Github GraphQL API v4 2017. https://developer.github.com/v4/. (2017).

[7] Github GraphQL API v4, GraphQL resource limitations 2017. https://developer.github.com/v4/guides/resource-limitations/. (2017).

[8] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2001. The complexity of acyclic conjunctive queries. *J. ACM* 48, 3 (2001), 431–498.

[9] GraphQL Users 2017. http://graphql.org/users/. (2017).

[10] Olaf Hartig and Jorge Pérez. 2017. An Initial Analysis of Facebook's GraphQL Language. In *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW).*

[11] D. S. Johnson. 1990. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, J. van Leeuwen (Ed.). Vol. A. Elsevier, Chapter 2.

[12] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013.* 297–308.

[13] Markus Kröll, Reinhard Pichler, and Sebastian Skritek. 2016. On the Complexity of Enumerating the Answers to Well-designed Pattern Trees. In *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016.* 22:1–22:18.

[14] Ivo Meißner. 2017. GraphQL Query Complexity Analysis for graphql-js. https://github.com/ivome/graphql-query-complexity/. (2017).

[15] Reinhard Pichler and Sebastian Skritek. 2013. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.* 79, 6 (2013), 984–1001.

[16] Leonard Richardson, Mike Amundsen, and Sam Ruby. 2013. *RESTful Web APIs.* O'Reilly Media, Inc.

[17] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013.* 10–20.

[18] Luc Segoufin and Alexandre Vigny. 2017. Constant Delay Enumeration for FO Queries over Databases with Local Bounded Expansion. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy.* 20:1–20:16.

[19] Stem Disintermedia, Inc. 2016. Join Monster. https://github.com/stems/join-monster. (2016).

[20] Stem Disintermedia, Inc. 2017. GraphQL Depth Limit. https://github.com/stems/graphql-depth-limit/. (2017).

[21] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA.* 137–146.

[22] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings.* 82–94.