An Initial Analysis of Facebook's GraphQL Language

Olaf Hartig[1] and Jorge Pérez[2]

[1] Dept. of Computer and Information Science (IDA), Linköping University, Sweden
olaf.hartig@liu.se

[2] Department of Computer Science, Universidad de Chile
jperez@dcc.uchile.cl

Abstract Facebook's 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. The goal of this paper is to understand the properties of this language. To this end, we first provide a formal query semantics. Thereafter, we analyze the language and show that it has a very low complexity for evaluation. More specifically, we show that the combined complexity of the main decision problems is in NL (Nondeterministic Logarithmic Space) and, thus, they can be solved in polynomial time and are highly parallelizable.

1 Introduction

After developing and using it internally for three years, in 2016, Facebook released a specification [2] and a reference implementation[3] 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 [6]. Since its release GraphQL has gained significant momentum and has been adopted by an increasing number of users.[4]

A core component of the GraphQL framework is a query language that is used to express the data retrieval requests issued to GraphQL-aware Web servers. While there already exist a number of implementations of this language (which is not to be confused with an earlier graph query language of the same name [5]), a more fundamental understanding of the properties of the language is missing. This paper presents preliminary work towards closing this gap, which is important to identify fundamental limitations and optimization opportunities of possible implementations and to compare the GraphQL framework to other Web-based data access interfaces such as REST services [6], SPARQL endpoints [3], and other Linked Data Fragments interfaces [8].

Before we describe the contributions of our work, we briefly sketch the idea of queries and querying in the GraphQL framework: Syntactically, GraphQL queries resemble the JavaScript Object Notation (JSON) [1]. However, in contrast to arbitrary JSON objects, GraphQL queries are written in terms of a so-called *schema* which the queried Web server supports [2]. Informally, such a schema defines types of objects by specifying a set of so-called *fields* for which the objects may have values; the possible values can be restricted to a specific type of scalars or objects. For instance, Figure 1 presents a GraphQL schema specification about Star Wars movies, which is also given in a JSON-like syntax [2] (we describe the elements in the schema specification in more detail below). Figure 2(a) presents a corresponding GraphQL query that, for the hero of

[3] http://graphql.org/code/

[4] http://graphql.org/users/ (note that popular sites such as Coursera, Github, Pinterest are users)

```
type Starship {
  id: ID!
  name: String!
  length(unit: String): Float
}

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

type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

```
type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

union SearchResult = Human | Droid | Starship

enum Episode { NEWHOPE, EMPIRE, JEDI }

type Query {
  hero(episode: Episode!): Character
  droid(id: ID!): Droid
  node(id: ID!): SearchResult
}
```

Figure 1. Example GraphQL schema (from `http://graphql.org/learn/schema/`).

the `JEDI` episode, returns the name and the episodes that the hero appears in; additionally, the query returns the value of either the `totalCredits` or the `primaryFunction` field, depending on whether the hero is a human or a droid. The GraphQL specification defines the semantics of such queries by using an operational definition that assumes an *"internal function [...] for determining the [...] value of [any possible] field"* of any given object [2]. This internal function is not specified any further and, instead, it is left to the implementation what exactly this function does. Hence, the given query semantics is not formally grounded in any specific data model. However, the data that is exposed via a GraphQL interface can be conceived of as a virtual, graph-based view of an underlying dataset; this view is established by the implementation of the aforementioned internal function and it takes the form of a graph that is similar to a Property Graph [7].

As a basis for our work we define a logical data model that formally captures the notion of this 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). These are the main conceptual contributions of the paper. As a technical contribution we use our formalization to study the computational complexity of the language (cf. Section 4). We show that, even though the size of query results may be exponential in the size of the queries, the evaluation decision problem lies in a very low complexity class; it can be solved in Nondeterministic Logarithmic Space and, thus, is highly parallelizable.

2 Data Model

The GraphQL specification does not provide a definition of a data model that is used as the foundation of GraphQL. However, the specification implicitly assumes a logical data model that is implemented as a virtual, graph-based view over some underlying DBMS. In this section we make this logical data model explicit by providing a formal definition thereof. For each concept of the GraphQL specification that our definitions capture, we refer to corresponding section of the specification that introduces the concept.

2.1 GraphQL Schema

We consider the following infinite countable sets: `Fields` (field names, §2.5 of [2]), `Arguments` (argument names, §2.6 of [2]), `Types` (type names, §3.1 of [2]) , and we assume that `Fields`, `Arguments` and `Types` are disjoint and that there exists a finite

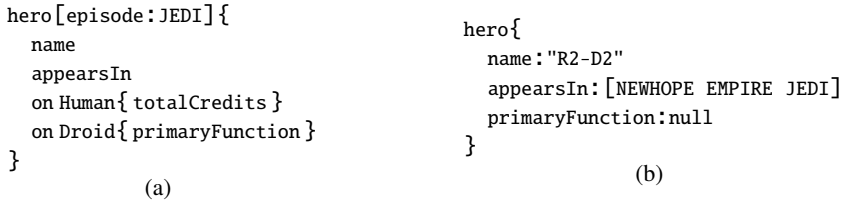

Figure 2. Example query and result.

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

GraphQL schemas and graphs are defined over finite subsets of the above sets. Let F, A, and T be finite sets such that F $\subset$ Fields, A $\subset$ Arguments, and T $\subset$ Types. We also assume that T is the disjoint union of $\texttt{O}_\texttt{T}$ (object types, §3.1.2 of [2]), $\texttt{I}_\texttt{T}$ (interface types, §3.1.3 of [2]), $\texttt{U}_\texttt{T}$ (union types, §3.1.4 of [2]) and Scalars, and we denote by $\texttt{L}_\texttt{T}$ the set $\{[\ \texttt{t}\ ] \mid \texttt{t} \in \texttt{T}\}$ of list types constructed from T (cf. §3.1.7 of [2]).

We now have all the necessary to define a GraphQL schema over (F, A, T).

Definition 1. *A GraphQL schema $\mathcal{S}$ over* (F, A, T) *is composed of three assignments:*

- $\mathit{fields}_\mathcal{S} : (\texttt{O}_\texttt{T} \cup \texttt{I}_\texttt{T}) \to 2^\texttt{F}$ *that assigns a set of fields to every object or interface type,*
- $\mathit{args}_\mathcal{S} : \texttt{F} \to 2^\texttt{A}$ *that assigns a set of arguments to every field,*
- $\mathit{type}_\mathcal{S} : \texttt{F} \cup \texttt{A} \to \texttt{T} \cup \texttt{L}$ *that assigns a type or a list type to every field and argument, where arguments are assigned scalar types; i.e.,* $\mathit{type}_\mathcal{S}(\texttt{a}) \in \texttt{Scalars}$ *for all* $\texttt{a} \in \texttt{A}$*,*

and functions that define interface and union types:

- $\mathit{union}_\mathcal{S} : \texttt{U}_\texttt{T} \to 2^{\texttt{O}_\texttt{T}}$ *that assigns a nonempty set of object types to every union type,*
- $\mathit{implementation}_\mathcal{S} : \texttt{I}_\texttt{T} \to 2^{\texttt{O}_\texttt{T}}$ *that assigns a set of object types to every interface.*

Additionally, $\mathcal{S}$ contains a distinguished type $q_\mathcal{S} \in \texttt{O}_\texttt{T}$ called the query type.

For the sake of avoiding an overly complex formalization, in our definition of a GraphQL schema we ignore the additional notions of *input types* (cf. §3.1.6 of [2]), *non-null types* (cf. §3.1.8 of [2]), and a *mutation type* (§3.3 of [2]). However, we capture the concept of interfaces and their implementation (cf. §3.1.3 of [2]) by introducing a notion of consistency. Informally, 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 $\mathit{fields}_\mathcal{S}(\texttt{i}) \subseteq \mathit{fields}_\mathcal{S}(\texttt{t})$ for every $\texttt{t} \in \mathit{implementation}_\mathcal{S}(\texttt{i})$. From now on we assume that all GraphQL schemas in this paper are consistent.

Example 1. The schema specified in Figure 1 may be captured as $\mathcal{S}$ over (F, A, T) with

$$\texttt{F} = \{\texttt{id}_f, \texttt{name}, \texttt{friends}, \texttt{appearsIn}, \texttt{starships}, \texttt{totalCredits}, \texttt{primaryFunction}, \texttt{length}, \texttt{hero}, \texttt{droid}, \texttt{node}\},$$

$$\texttt{A} = \{\texttt{id}_a, \texttt{unit}, \texttt{episode}\}, \text{ and}$$

$$\texttt{T} = \texttt{O}_\texttt{T} \cup \texttt{I}_\texttt{T} \cup \texttt{U}_\texttt{T} \cup \texttt{Scalars} \text{ such that:}$$

$$\texttt{O}_\texttt{T} = \{\texttt{Human}, \texttt{Droid}, \texttt{Starship}, \texttt{Query}\}, \qquad \texttt{I}_\texttt{T} = \{\texttt{Character}\},$$

$$\texttt{Scalars} = \{\texttt{ID}, \texttt{String}, \texttt{Int}, \texttt{Float}, \texttt{Episode}\}, \qquad \texttt{U}_\texttt{T} = \{\texttt{SearchResult}\}.$$

[5] For the sake of simplicity, we assume that Scalars includes the *enum types* that are treated separately in the GraphQL specification (cf. §3.1.5 of [2]).

As we can see in Figure 1, in the original GraphQL syntax, object types are defined using the keyword `type`, interface types with the keyword `interface`, and union types with the keyword `union`. Also notice that the schema in Figure 1 introduces both a field and an argument with name `id`. To avoid ambiguities we have used two different names: $\texttt{id}_f$ and $\texttt{id}_a$. The values for the scalar types are implicit in their names (`String`, `Float`, `Int`) except for `ID` which is a special scalar type used for unique identifiers in a GraphQL schema specification (cf. §3.1.1.5 of [2]), and `Episodes` which is an *enum type* such that $\mathit{values}(\texttt{Episodes}) = \{\texttt{NEWHOPE}, \texttt{EMPIRE}, \texttt{JEDI}\}$. Regarding the functions that compose $\mathcal{S}$ we have that $\mathit{fields}_{\mathcal{S}}$ defines the assignments:

$$\begin{aligned}
\texttt{Starship} &\to \{\texttt{id}_f, \texttt{name}, \texttt{length}\},\\
\texttt{Character} &\to \{\texttt{id}_f, \texttt{name}, \texttt{friends}, \texttt{appearsIn}\},\\
\texttt{Droid} &\to \{\texttt{id}_f, \texttt{name}, \texttt{friends}, \texttt{appearsIn}, \texttt{primaryFunction}\},\\
\texttt{Human} &\to \{\texttt{id}_f, \texttt{name}, \texttt{friends}, \texttt{appearsIn}, \texttt{starships}, \texttt{totalCredits}\},\\
\texttt{Query} &\to \{\texttt{hero}, \texttt{droid}, \texttt{node}\}.
\end{aligned}$$

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

$$\begin{aligned}
\texttt{length} &\to \{\texttt{unit}\}, \quad \texttt{hero} \to \{\texttt{episode}\},\\
\texttt{droid} &\to \{\texttt{id}_a\}, \quad \texttt{node} \to \{\texttt{id}_a\},
\end{aligned}$$

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

$$\begin{aligned}
\texttt{id}_a &\to \texttt{ID}, & \texttt{episode} &\to \texttt{Episode}, & \texttt{unit} &\to \texttt{String},\\
\texttt{id}_f &\to \texttt{ID}, & \texttt{friends} &\to [\texttt{Character}], & \texttt{name} &\to \texttt{String},\\
\texttt{droid} &\to \texttt{Droid}, & \texttt{appearsIn} &\to [\texttt{Episode}], & \texttt{hero} &\to \texttt{Character},\\
& & \texttt{totalCredits} &\to \texttt{Int}, & \texttt{node} &\to \texttt{SearchResult},\\
& & \texttt{primaryFunction} &\to \texttt{String}, & \texttt{length} &\to \texttt{Float},
\end{aligned}$$

The functions that define union and interface types, respectively, are such that

$$\begin{aligned}
\mathit{union}_{\mathcal{S}}(\texttt{SearchResult}) &= \{\texttt{Human}, \texttt{Droid}, \texttt{Starship}\},\\
\mathit{implementation}_{\mathcal{S}}(\texttt{Character}) &= \{\texttt{Human}, \texttt{Droid}\}.
\end{aligned}$$

2.2 GraphQL Graphs

The logical data model assumed by GraphQL considers data that can be represented in a graph-based form. Such a graph is a directed, edge-labeled multigraph in which each node has a type and properties. We define this graph by using the aforementioned domain (`F`, `A`, `T`). Then, each node in the graph is associated with an object type from `T`. The edge labels, as well as the names of node properties, consist of a field name from `F` and a set of arguments, where such an argument is a pair consisting of a distinct argument name from `A` and a corresponding value (note that the set of arguments may be empty). The value of each node property is either a single scalar value or a sequence thereof. The following definition captures our notion of a GraphQL graph formally.

Definition 2. *A* GraphQL graph *over* (`F`, `A`, `T`) *is a tuple* $G = (N, E, \tau, \lambda)$ *where:*

- N *is a set of nodes,*
- E *is a set of edges of the form* $(u, \texttt{f}[\alpha], v)$ *where* $u, v \in N$, $\texttt{f} \in \texttt{F}$, *and* α *is a partial mapping from* `A` *to* `Vals`,
- $\tau : N \to \texttt{O}_\texttt{T}$ *is a total function that assigns a type to every node, and*

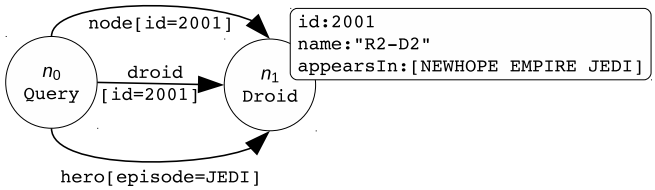

Figure 3. Example GraphQL graph.

– *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* $\mathtt{A}$ *to* $\mathtt{Vals}$*.*

Example 2. Figure 3 illustrates a small GraphQL graph $G_{\mathsf{ex}} = (N_{\mathsf{ex}}, E_{\mathsf{ex}}, \tau_{\mathsf{ex}}, \lambda_{\mathsf{ex}})$ over the domain $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ as given in Example 1. G_{ex} contains two nodes, $N_{\mathsf{ex}} = \{n_0, n_1\}$, and three edges, including $(n_0, \mathtt{droid}[\alpha_1], n_1) \in E_{\mathsf{ex}}$ with $\alpha_1(\mathtt{id}_a) = 2001$. Function τ_{ex} defines the assignments:

$$n_0 \to \mathtt{Query}, \qquad n_1 \to \mathtt{Droid},$$

and function λ_{ex} defines the assignments:

$$(n_1, \mathtt{id}_f[\alpha_\emptyset]) \to \mathtt{2001}, \qquad (n_1, \mathtt{name}[\alpha_\emptyset]) \to \mathtt{"R2\text{-}D2"},$$
$$(n_1, \mathtt{appearsIn}[\alpha_\emptyset]) \to [\mathtt{NEWHOPE\ EMPIRE\ JEDI}].$$

Observe that Definition 2 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 that conformance to a schema imposes on a GraphQL graph are summarized as follows: For every edge, the field name that labels the edge is among the field names that the schema specifies for the type of the source node of the edge. The type that the schema associates with this field name must match the type of the target node of the edge, and if this type associated with the field name is not a list type, then the target node is the only node connected to the source node by an edge with the given field name. Moreover, for every argument associated with an edge, the argument name must be among the argument names that the schema associates with the field name that labels the edge, and the value of the argument must be of the type associated with that argument name. In addition to these conditions for the edge labels, there exist similar conditions for the node properties. Finally, the graph must contain a designated node whose type is the query type of the schema. Providing a formal definition of these conditions is straightforward. Due to space limitations, we therefore omit the definition in this paper.

Example 3. The GraphQL graph in Example 2 conforms to the schema in Example 1.

3 Definition of the GraphQL Query Language

In this section we provide a formal definition of the GraphQL query language. In particular, we first define a concise syntax of GraphQL queries that resembles closely the JSON-like syntax introduced in the GraphQL specification (cf. §2 of [2]). Thereafter, we define a formal semantics of these queries. However, 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 over a GraphQL graph.

The most basic construction in our syntax of GraphQL queries are expressions of the form $\mathtt{f}[\alpha]$. Informally, when evaluated over a GraphQL 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 $\mathtt{v}$, or a sequence $[\mathtt{v}_1 \cdots \mathtt{v}_n]$ of scalar values, then the result of the evaluation is a string of the form $\mathtt{f{:}v}$, or $\mathtt{f{:}}[\mathtt{v}_1 \cdots \mathtt{v}_n]$. An alternative to the construction $\mathtt{f}[\alpha]$ is $\ell\mathtt{:f}[\alpha]$ which captures the notion of *"field aliases"* (cf. §2.7 of [2]). Such aliases can be used to rename the field names that appear in the query result. That is, when using this alternative construction, the results will be strings of the form $\ell\mathtt{:v}$ or $\ell\mathtt{:}[\mathtt{v}_1 \cdots \mathtt{v}_n]$.

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: $\ell\mathtt{:f}[\alpha]\{\varphi\}$.

Our query syntax introduces two more constructions: $\mathtt{on\ t}\{\varphi\}$ 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 of [2]). 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 non-terminal nodes whose associated (object) type is compatible with $\mathtt{t}$ (a formal definition of this notion of compatibility follows shortly).

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 of [2]), *"variables"* (§2.10 of [2]), and *"directives"* (§2.12 of [2]). 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. *A* GraphQL query *over* $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ *is an expression constructed from the following grammar where* $\mathtt{[}$, $\mathtt{]}$, $\mathtt{\{}$, $\mathtt{\}}$, $\mathtt{:}$, *and* $\mathtt{on}$ *are terminal symbols,* $\mathtt{f} \in \mathtt{F}$, $\ell \in \mathtt{Fields}$, $\mathtt{t} \in \mathtt{O_T} \cup \mathtt{I_T} \cup \mathtt{U_T}$, *and* α *represents a partial mapping from* $\mathtt{A}$ *to* $\mathtt{Vals}$.

$$\varphi ::= \ \mathtt{f}[\alpha] \ \mid \ \ell\mathtt{:f}[\alpha] \ \mid \ \mathtt{f}[\alpha]\{\varphi\} \ \mid \ \ell\mathtt{:f}[\alpha]\{\varphi\} \ \mid \ \mathtt{on\ t}\{\varphi\} \ \mid \ \varphi \cdots \varphi$$

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 $\mathtt{f}$.

To define a formal semantics of GraphQL queries we first observe that the query semantics described in the GraphQL specification assume that queries satisfy a notion of validity w.r.t. a given GraphQL schema (cf. §5 of [2]). We make the same assumption. To this end, we define validity of queries in terms of our formalization as follows.

Definition 4. *Let* $\mathcal{S}$ *be a GraphQL schema over* $(\mathtt{F}, \mathtt{A}, \mathtt{T})$, *let* Q *be a GraphQL query over* $(\mathtt{F}, \mathtt{A}, \mathtt{T})$, *and let* $\mathtt{t}^*$ *be a type in* $\mathtt{O_T} \cup \mathtt{I_T} \cup \mathtt{U_T} \subseteq \mathtt{T}$. *Then,* Q conforms *to* $\mathcal{S}$ *in the context of* $\mathtt{t}^*$, *denoted by* $Q \models^{\mathtt{t}^*} \mathcal{S}$, *if* Q *satisfies the following conditions:*

1. *If* Q *is of the form* $\mathtt{f}[\alpha]$ *or* $\ell\mathtt{:f}[\alpha]$, *then*
   - $\mathtt{t}^* \notin \mathtt{U_T}$, $\mathtt{f} \in \mathit{fields}_{\mathcal{S}}(\mathtt{t}^*)$, $\mathrm{dom}(\alpha) \subseteq \mathit{args}_{\mathcal{S}}(\mathtt{f})$, *and*
   - *assuming* $\mathit{type}_{\mathcal{S}}(\mathtt{f}) = \mathtt{t}$ *or* $\mathit{type}_{\mathcal{S}}(\mathtt{f}) = [\mathtt{t}]$, $\mathtt{t} \in \mathtt{Scalars}$.
2. *If* Q *is of the form* $\mathtt{f}[\alpha]\{\varphi\}$ *or* $\ell\mathtt{:f}[\alpha]\{\varphi\}$, *then*
   - $\mathtt{t}^* \notin \mathtt{U_T}$, $\mathtt{f} \in \mathit{fields}_{\mathcal{S}}(\mathtt{t}^*)$, $\mathrm{dom}(\alpha) \subseteq \mathit{args}_{\mathcal{S}}(\mathtt{f})$, *and*

– *assuming* $type_{\mathcal{S}}(\mathtt{f}) = \mathtt{t}$ *or* $type_{\mathcal{S}}(\mathtt{f}) = [\mathtt{t}]$, $\mathtt{t} \notin \mathtt{Scalars}$ *and* $\varphi \models^{\mathtt{t}} \mathcal{S}$.

3. *If Q is of the form* $\mathtt{on\ t}\{\varphi\}$, *then* $\varphi \models^{\mathtt{t}} \mathcal{S}$.
4. *If Q is of the form $\varphi_1 \cdots \varphi_n$, then $\varphi_i \models^{\mathtt{t}^*} \mathcal{S}$ for every $\varphi_i \in \{\varphi_1, \ldots, \varphi_n\}$.*

Moreover, we say that Q conforms *to $\mathcal{S}$ if $Q \models^{q_S} \mathcal{S}$ (where q_S is the query type of $\mathcal{S}$).*

Example 4. The GraphQL query in Figure 2(a) conforms to the schema in Example 1.

As a last preliminary for formalizing the query semantics of GraphQL we require a definition of the notion of a result that a GraphQL query may return. As for the queries, we use expressions that resemble the expressions used in the GraphQL specification.

Definition 5. *A* GraphQL result object *is constructed from the following grammar where $\ell \in \mathtt{Fields}$, $\mathtt{v}, \mathtt{v}_1, \ldots, \mathtt{v}_n \in \mathtt{Vals}$, and* `{`, `}`, `[`, `]`, `:`, *and* `null` *are terminal symbols:*

$$\rho ::= \quad \ell\mathtt{:v} \;\mid\; \ell\mathtt{:[v}_1 \cdots \mathtt{v}_n\mathtt{]} \;\mid\; \ell\mathtt{:\{}\rho\mathtt{\}} \;\mid\; \ell\mathtt{:[\{}\rho\mathtt{\}} \cdots \mathtt{\{}\rho\mathtt{\}]} \;\mid\; \rho \cdots \rho \;\mid\; \ell\mathtt{:null}$$

We now are ready to define a formal semantics of GraphQL queries. To this end, we introduce an evaluation function that, for any GraphQL query and any GraphQL graph (both conforming to a given schema), defines the corresponding query result.

Definition 6. *Let $G = (N, E, \tau, \lambda)$ be a GraphQL graph over* $(\mathtt{F}, \mathtt{A}, \mathtt{T})$, *let Q be a GraphQL query over* $(\mathtt{F}, \mathtt{A}, \mathtt{T})$, *and let $\mathcal{S}$ be a GraphQL schema over* $(\mathtt{F}, \mathtt{A}, \mathtt{T})$ *such that G and Q conform to $\mathcal{S}$. The $\mathcal{S}$-specific evaluation of Q over G from node $u \in N$, denoted by $[\![Q]\!]^u_G$, is a GraphQL result object that is defined recursively as follows.*

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

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

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

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

$$[\![\mathtt{on\ t}\{\varphi\}]\!]^u_G = \begin{cases} [\![\varphi]\!]^u_G & \text{if } \mathtt{t} \in \mathtt{O_T} \text{ and } \tau(u) = \mathtt{t} \\ [\![\varphi]\!]^u_G & \text{if } \mathtt{t} \in \mathtt{I_T} \text{ and } \tau(u) \in implementation_{\mathcal{S}}(\mathtt{t}) \\ [\![\varphi]\!]^u_G & \text{if } \mathtt{t} \in \mathtt{U_T} \text{ and } \tau(u) \in union_{\mathcal{S}}(\mathtt{t}) \\ \varepsilon & \text{in other case. } (\varepsilon \text{ denotes the empty word}) \end{cases}$$

$$[\![\varphi_1 \cdots \varphi_k]\!]^u_G = [\![\varphi_1]\!]^u_G \cdots [\![\varphi_k]\!]^u_G$$

Finally, the $\mathcal{S}$-specific evaluation of Q over G, denoted by $[\![\varphi]\!]_G$, is simply $[\![\varphi]\!]_G = [\![\varphi]\!]^u_G$ where u is the single node in G such that $\tau(u) = q_{\mathcal{S}}$.

In the third and fourth cases in Definition 6 whenver $type_{\mathcal{S}}(\mathtt{f}) \in \mathtt{L_T}$ the evaluation produces a sequence from a set of nodes $\{v_1, \ldots, v_k\}$. Notice that the order of the sequence depends on the order in which we consider the nodes in the previous set. The

original GraphQL specification implicitly assumes an order associated to every outgoing edge in the graph, and this order is used to produce the mentioned sequences. To keep our formalization as simple as possible we did not formally introduce these order relations in this paper.

Example 5. Consider the GraphQL query in Figure 2(a) and the GraphQL schema $\mathcal{S}$ introduced in Example 1. Figure 2(b) illustrates the result of the $\mathcal{S}$-specific evaluation of this example query over the graph in Example 2.

4 Complexity Results

Notice that a GraphQL query has as solution always a single result object, nevertheless, this result may be of exponential size as stated in the following proposition.

Proposition 1. *For every GraphQL query φ and GraphQL graph G it holds that $[\![\varphi]\!]_G$ is of size $O(|G|^{|\varphi|})$. Moreover, there exists a family of queries $\{\varphi_n\}_{n\geq 1}$ and a graph G, such that every query φ_n is of size $O(n)$ but the evaluation $[\![\varphi_n]\!]_G$ is of size $\Omega(2^n)$.*

Proof (sketch). Consider a schema with an object type `Person` with fields `name` and `knows` such that `name` is a scalar, and `knows` is a list in which every element is of type `Person`. Additionally, the query type $q_{\mathcal{S}}$ has a single field `query` of type `Person`. Let $G = (V, E, \tau, \lambda)$ be such that $V = \{u, v, w, x\}$, $E = \{(u, \texttt{query}, v), (v, \texttt{knows}, w), (v, \texttt{knows}, x), (w, \texttt{knows}, v), (x, \texttt{knows}, v)\}$, $\tau(u) = q_{\mathcal{S}}$, $\tau(v) = \tau(w) = \tau(x) = \texttt{Person}$, $\lambda(v, \texttt{name}) = \texttt{Alice}$. Consider now the queries given by the following recurrence

$$\alpha_1 = \texttt{name}, \qquad \text{and} \qquad \alpha_i = \texttt{knows\{knows\{}\alpha_{i-1}\texttt{\}\}} \quad (\text{for every } i > 1),$$

and define φ_n as `query{`α_n`}`. It is clear that the size of φ_n is linear in n but the evaluation of φ_n over G is of size exponential in n; in fact, the name `Alice` occurs 2^{n-1} times in $[\![\varphi_n]\!]_G$. The exponential upper bound can be proved by a simple induction argument over the evaluation function in Definition 6. □

We now go to some of the related decision problems for GraphQL. Notice that the result of a GraphQL query is not a set of tuples (as it is for more classical query languages). Thus, we need to introduce some further notions to properly define the evaluation decision problem in our context. Given two GraphQL result objects ρ and ρ', we say that ρ *occurs in* ρ' if ρ is a substring of ρ'. Notice that a result object may occur several times in another. We next introduce the notion of *removing* a result object. Assume that ρ occurs in ρ', then the result object obtained by removing ρ from ρ' is the substring of ρ' obtained from deleting an (arbitrary) occurrence of ρ and then recursively performing the following procedure:

1. delete every substring of the form `{}`,
2. delete every substring of the form ℓ`:[]` (with $\ell \in$ `Fields`),
3. repeat the two above steps until no further deletion is possible.

The above procedure ensures that removing a result object from another produces a valid result object. For instance, consider the following result objects:

ρ_1 = `p:{n:Alice knows:[{n:Bob}{n:Charly}] son:{n:Dylan knows:[{n:Ed}]}}`
ρ_2 = `n:Dylan knows:[{n:Ed}]`
ρ_3 = `p:{n:Alice knows:[{n:Bob}{n:Charly}]}`
ρ_4 = `p:{n:Alice knows:[{n:Bob}]}`
ρ_5 = `knows:[{n:Bob}]`

Then ρ_2 occurs in ρ_1. Also notice that, although ρ_3 does not occur in ρ_1, result object ρ_3 is obtained from ρ_1 by removing ρ_2. Now, we say that ρ *is a reduction of* ρ' if ρ can be obtained from ρ' by removing some result objects that occur in it. Finally, we say that ρ *is a subresult of* ρ' if ρ occurs in a reduction of ρ'. Considering our example above, we have that ρ_4 is a reduction of ρ_3 and thus is also a reduction of ρ_1. Moreover, since ρ_5 occurs in ρ_4, we have that ρ_5 is a subresult of ρ_1. Intuitively, when ρ is a subresult of ρ', we know that all the data in ρ appears in ρ' respecting the structure of the original result object. With these definitions we can formalize the following decision problem:

Problem	: GqlEval
Input	: a GraphQL query φ, a graph G, and a result object ρ
Question	: is ρ a subresult of $[\![\varphi]\!]_G$?

Theorem 1. GqlEval *is NL-complete.*

Proof (Sketch). One main ingredient in the proof is to see GraphQL queries and result objects as trees. Intuitively, a query can be seen as a edge-labeled tree that follows the structure of the `{}` symbols. For instance the GraphQL query `a{b{c d} e{f}}` can be represented as a tree in which the root has a single outgoing `a`-labeled edge to a node, say n, with two outgoing edges labeled with `b` and `e`. The `b`-child of n has two outgoing edges labeled with `c` and `d`, while the `e`-child of n has only one `f`-labeled outgoing edge. For result objects the situations is similar but structures of the form `a:[`$\{\rho_1\}\cdots\{\rho_k\}$`]` represents several `a`-labeled edges to every one of the trees constructed from $\rho_1, \ldots, \rho_k$, and structures of the form `a:v` with `v` a scalar, represent an edge pointing to a leaf node labeled with `v`. With this representation we can talk about root, paths, and leaves in GraphQL queries and result objects, respectively. Another observation is that we 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, by using logarithmic space. A similar traversal can be done for result objects.

We now have all the necessary to sketch the NL membership of GqlEval. First we guess the position, say p, of a label in φ, and a node, say u, in G. The intuition is that p represents the part of the query φ that when evaluated over G from node u contains ρ as a subresult object. This last property can be checked in NL as follows. We first check that ρ matches the schema of query φ at position p. To this end, we consider every edge of the form `a:v` (that is, an edge to a leaf node) in ρ and its corresponding path from the root of ρ. Lets denote by $P_{\texttt{a:v}}$ this path (which is essentially a sequence of labels). Then we check that there is a path in the query φ starting at p that matches $P_{\texttt{a:v}}$. Notice that we are performing reachability tests that can de done in NL (actually in L since the reachability is on trees). We still need to check two further properties: (1) that node u can be reached from the query node in G by following the corresponding path in φ from the root to position p, and that (2) the data in result object ρ can actually be obtained

from G starting at node u. Check (1) can be done in NL by a standard reachability test. To check (2) we only need to iterate over all leaves of the form `a:v` in ρ and check that there exists a path in G from u to a node v that matches the labels of the path from the root of ρ to `a:v`, and such that $\lambda(v, a) = \mathtt{v}$. It can be proven that these checks are a necessary and sufficient condition to check that ρ is a subresult of $[\![\varphi]\!]_G$.

NL-hardness follows easily from the reachability problem in directed graphs. Given a directed graph G with N nodes and two nodes u and v, we create a GraphQL graph G' by adding a label, say `a` to every edge, an initial query node q to G, an edge labeled `q` from q to u, and a data value, say `1`, associated to attribute `b` in node v. Types of nodes in G' are arbitrary. Then we consider the sequence of queries constructed recursively as $\alpha_0 = \mathtt{b}$ and $\alpha_i = \mathtt{b\ a\{}\alpha_{i-1}\mathtt{\}}$, and the query $\varphi = \mathtt{q\{}\alpha_{N-1}\mathtt{\}}$. It is not difficult to argue that G' and φ can be constructed from G by using logarithmic space. Moreover, the result object `b:1` is a subresult of $[\![\varphi]\!]_{G'}$ if and only if v is reachable from u in G. □

5 Concluding Remarks and Future Work

We have embarked on the study of Facebook's GraphQL language; that is, we have formalized its syntax and semantics and presented some initial complexity results. This new language opens several interesting directions for future research.

Our current ongoing work includes a complexity study and algorithm design for more practical problems related to GraphQL, including the parallel evaluation of queries and the estimation of the size of a result object before computing the result. Our initial findings show that, as for the case of the evaluation decision problems, these problems also have a very low computational complexity.

As another important topic for future research we plan to compare the GraphQL query language with classical query languages. An immediate candidate in terms of expressive power and complexity is the language of acyclic conjunctive queries (ACQs). The (combined) complexity of ACQs is LOGCFL-complete [4] and, since it is believed that NL $\subsetneq$ LOGCFL, the membership of GqlEval in NL shows an important difference in terms of complexity between the two languages.

References

1. ECMA. *ECMA-404: The JSON Data Interchange Format*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, 2013.
2. Facebook, Inc. GraphQL. Working Draft, Oct. 2016. Online at `http://facebook.github.io/graphql`, retrieved on Dec. 12, 2016.
3. L. Feigenbaum and G. T. Williams. SPARQL 1.1 Protocol for RDF. W3C Rec., 2013.
4. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.
5. H. He and A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 405–418, 2008.
6. L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013.
7. I. Robinson, J. Webber, and E. Eifrém. *Graph Databases*. O'Reilly Media, 2013.
8. R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. *Journal of Web Semantics*, 37–38:184–206, 2016.