Native Execution of GraphQL Queries over RDF Graphs Using Multi-Way Joins

Nikolaos KARALIS a,1, Alexander BIGERL a and Axel-Cyrille NGONGA NGOMO a
a DICE group, Department of Computer Science, Paderborn University
ORCiD ID: Nikolaos Karalis https://orcid.org/0000-0002-0710-7180, Alexander Bigerl
https://orcid.org/0000-0002-9617-1466, Axel-Cyrille Ngonga Ngomo
https://orcid.org/0000-0001-7112-3516

Abstract. Purpose: The query language GraphQL has gained significant traction in recent years. In particular, it has recently gained the attention of the semantic web and graph database communities and is now often used as a means to query knowledge graphs. Most of the storage solutions that support GraphQL rely on a translation layer to map the said language to another query language that they support natively, for example SPARQL. Methodology: Our main innovation is a multi-way left-join algorithm inspired by worst-case optimal multi-way join algorithms. This novel algorithm enables the native execution of GraphQL queries over RDF knowledge graphs. We evaluate our approach in two settings using the LinGBM benchmark generator. Findings: The experimental results suggest that our solution outperforms the state-of-the-art graph storage solution for GraphQL with respect to both query runtimes and scalability. Value: Our solution is implemented in an open-sourced triple store, and is intended to advance the development of representation-agnostic storage solutions for knowledge graphs.

Keywords. graphql, knowledge graphs, multi-way joins

1. Introduction

Knowledge graphs serve as the data backbone of an increasing number of applications. Examples of such applications include search engines, recommendation systems, and question answering systems [1,2]. Consequently, efficient storage and querying solutions for knowledge graphs are imperative. Many triple stores [3,4,5,6,7,8,9,10] and graph databases [11,12] have hence been developed in recent decades. Used primarily by the semantic web community, triple stores process RDF knowledge graphs. A popular representation model among the graph database community is the property graph model [2,13]. While SPARQL is the designated query language for RDF, multiple languages have been developed to query property graphs (e.g., Cypher [14] and Gremlin [15]). Re-
recently, GraphQL, a query language for APIs, has attracted the attention of both the graph database [16,17] and the semantic web [18,19,20,21] communities.

The focus of the semantic web community regarding GraphQL has been on the development of query translation tools [18,20,21]. These tools translate GraphQL queries into SPARQL queries, which are then issued to a triple store. A drawback of such solutions is that the results produced by triple stores need to be rewritten, since GraphQL dictates a strict response format. This process adds a significant overhead to the execution of queries [18, Table 3], especially in cases where the results are large. To the best of our knowledge, there are no publicly available triple stores that treat GraphQL as a first-class citizen.

While most constructs for processing basic graph patterns can be exploited in a straightforward manner for GraphQL processing, the formal semantics of GraphQL [22] demand the use of left-join operations for the evaluation of GraphQL queries. However, conventional two-way left-join operations are not suitable for the evaluation of GraphQL queries, as the results of such queries can be constructed incrementally [22]. We hence focus on presenting a novel multi-way left-join algorithm inspired by worst-case optimal join algorithms [23], which can be used to enumerate GraphQL queries incrementally. By implementing our approach into a state-of-the-art triple store, we provide the first publicly available triple store that treats GraphQL queries as first-class citizens.

We carried out an extensive evaluation using a synthetic benchmark generator, namely LinGBM [24], and the results suggest that our implementation is able to outperform a state-of-the-art graph storage solution providing GraphQL support, namely Neo4j.

The rest of this paper is structured as follows. The preliminaries are provided in Section 2. In Section 3, we present our multi-way left-join algorithm and show how to natively evaluate GraphQL queries over RDF graphs. We evaluate our approach in Section 4. We discuss related works in Section 5 and conclude in Section 6.

2. Preliminaries

Here, we introduce the concepts and the semantics of GraphQL that we use throughout this work along with their formal definitions as per [22]. We also briefly introduce worst-case optimal multi-way join algorithms, which have inspired our proposed algorithm.

2.1. GraphQL

GraphQL is a query language that was designed to simplify communication between clients and application servers. One of the main characteristics of GraphQL is that it is strongly typed. GraphQL services—i.e., servers and data sources whose data can be accessed and modified via GraphQL operations—expose a GraphQL schema to their clients by which incoming requests must abide. This schema defines a type system that describes the structure of the underlying data of the GraphQL service and the operations the service supports. Another important aspect of GraphQL is the hierarchical structure of its operations and responses. GraphQL operations form a tree structure that specifies the traversal on top of the underlying graph and the information that should be extracted from the nodes at each step of the traversal. In turn, the responses should follow the hierarchy defined by their respective operation. The syntax and capabilities of GraphQL,
as well as the responsibilities of GraphQL services, are detailed in the language’s official specification [25]. Even though the specification describes how services should handle the requests they receive, it does not provide a formal specification of the semantics of the language. Consequently, studying the expressiveness and complexity of the language remained a challenge. To tackle the lack of formal semantics and the consequences thereof, Hartig and Pérez [22] provide formal semantics for GraphQL queries that consist of fields, field aliases and inline fragments. The semantics rely on the formal definition of GraphQL schemata and graphs as well as the formalized syntax of GraphQL queries. Here, we reintroduce the definitions presented in [22].

The formal definitions presented below rely on the following sets. Let $\text{Fields}$ be an infinite set of field names and $F \subset \text{Fields}$ a finite subset of $\text{Fields}$. Let $A$ and $T$ be finite sets of argument names and type names, respectively, where $T$ is the disjoint union of $\text{OT}$ (object type names), $\text{IT}$ (interface type names), $\text{UT}$ (union type names) and $\text{Scalars}$ (scalar type names). Last, let $LT = \{[t] | t \in T\}$ be the set of list types constructed from $T$ and $\text{Vals}$ be a set of scalar values.

**Definition 2.1 (GraphQL schema [22])** A GraphQL schema $S$ over $(F, A, T)$ is composed of the following components:

- $\text{fields}_S : \text{OT} \cup \text{IT} \to 2^F$ that assigns a set of fields to every object type and every interface type,
- $\text{args}_S : F \to 2^A$ that assigns a set of arguments to every field,
- $\text{type}_S : F \cup A \to T \cup LT$ that assigns a type or a list type to every field and argument, where arguments are assigned scalar types,
- $\text{union}_S : \text{UT} \to 2^{\text{OT}}$ that assigns a nonempty set of object types to every union type,
- $\text{impl}_S : \text{IT} \to 2^{\text{OT}}$ that assigns a set of object types to every interface,
- $\text{root}_S \in \text{OT}$ that represents the query root type.

**Example 2.1** Consider the following GraphQL schema $S$

```graphql
interface Entity {
  id: String
  email: String
}
type Company impl Entity {
  id: String
  name: String
}
type Person impl Entity {
  id: String
  fname: String
  lname: String
  email: String
  age: Int
}
type Person impl Entity {
  employees:[Person]
}
type Query {
  people(lname:String):[Person]
  companies:[Company]
}
schema { query:Query }
```

Let $F = \{ \text{people, companies, employees, fname, age, id, lname, email, name} \}$, $A = \{ \text{lname} \}$, $\text{Or} = \{ \text{Query, Company, Person} \}$, $\text{Ir} = \{ \text{Entity} \}$, $\text{U}_T = \{ \}$, and $\text{Scalars} = \{ \text{String, Int} \}$. The GraphQL schema $S$ is formalized over $(F, A, T)$ as follows (we omit several assignments for brevity):

- $\text{args}_S(\text{people}) = \{\text{lname}\}$,
- $\text{fields}_S(\text{Entity}) = \{\text{id, email}\}$,
- $\text{type}_S(\text{id}) = \text{String}$,
- $\text{fields}_S(\text{Person}) = \{\text{id, fname, lname, email, age}\}$,
- $\text{root}_S = \text{Query}$. 
In practice, GraphQL services are implemented on top of data sources that adopt different data models (e.g., relational databases and graph databases). To provide the semantics of GraphQL queries in a unified manner, Hartig and Pérez [22] introduced the notion of GraphQL graphs. GraphQL graphs are logical constructs that abstract the underlying data sources of GraphQL services.

Definition 2.2 (GraphQL graph [22]) A GraphQL graph over \((F, A, T)\) is a tuple \(G = (N, E, \tau, \lambda, \rho)\) with the following elements:

- \(N\) is a set of nodes,
- \(E\) is a set of edges of the form \((u, f[a], v)\), where \(u, v \in N\), \(f \in F\), and \(a\) is a partial mapping from \(A\) to \(\text{Vals}\),
- \(\tau : N \rightarrow \mathcal{O}_T\) is a function that assigns a type to every node,
- \(\lambda\) is a partial function that assigns a scalar value \(\nu \in \text{Vals}\) or a sequence \([\nu_1 \ldots \nu_n]\) of scalar values (\(\nu_i \in \text{Vals}\)) to some pairs of the form \((u, f[a])\) where \(u \in N\), \(f \in F\) and \(a\) is a partial function from \(A\) to \(\text{Vals}\),
- \(\rho \in N\) is a distinguished node called the root node.

Definition 2.3 (GraphQL query [22]) A GraphQL query over \((F, A, T)\) is an expression \(\phi\) constructed from the following grammar where \(\ell, \mathcal{J}, \{\}, :\) and \(\on\) are terminal symbols, \(t \in \mathcal{O}_T \cup \mathcal{T} \cup \mathcal{U}_T\), \(f \in F\), \(\ell \in \text{Fields}\), and \(\alpha\) is a partial mapping from \(A\) to \(\text{Vals}\):

\[
\phi ::= f[\alpha] | \ell : f[\alpha] | \on t\{\phi\} | f[\alpha]\{\phi\} | \ell : f[\alpha]\{\phi\} | \phi \ldots \phi.
\]

Example 2.2 Examples of GraphQL queries conforming to the GraphQL schema \(S\) of Example 2.1 are the following:

\[
\phi_1 = \text{people}(lname: \text{"Doe"}) \{ \text{fname email} \} \text{ and } \\
\phi_2 = \text{companies} \{ \text{name employees} \{ \text{id lname} \} \}.
\]

Both queries demonstrate the hierarchical structure of GraphQL queries. For example, \(\phi_2\) accesses fields in the first level that belong to the object type \(\text{Company}\). In the second level, it accesses fields of the object type \(\text{Person}\), as \(\text{type}_{S} (\text{employees}) = [\text{Person}]\).

GraphQL queries of particular interest are those that are non-redundant and in ground-typed normal form. According to [22, Theorem 3.8], every GraphQL query can be transformed into an equivalent query that is non-redundant and in ground-typed normal form. An important characteristic of such queries is that their response can be constructed without being subjected to field collection [25, Section 6.3.2]. This allows non-redundant GraphQL queries in ground-typed normal form to be evaluated in time linear to the size of their response [22, Corollary 4.3].

Definition 2.4 (GraphQL semantics [22]) Let \(G = (N, E, \tau, \lambda, \rho)\) be a GraphQL graph and \(\phi\) a non-redundant GraphQL query in ground-typed normal form, both conforming to a GraphQL schema \(S\) over \((F, A, T)\). The evaluation of \(\phi\) over \(G\) from node \(u \in N\), denoted by \(\llbracket\phi\rrbracket^\rho_G\), is captured by Equation 1.\(^2\) The evaluation of \(\phi\) over \(G\), denoted by \(\llbracket\phi\rrbracket^\rho_G\), is simply \(\llbracket\phi\rrbracket^\rho_G\).

\(^2\) The expressions \(\ell : f[\alpha]\{\phi\}\) and \(\ell : f[\alpha]\) are evaluated by replacing \(f\) with \(\ell\) in the first two rules’ results.
\[ f_\alpha = \begin{cases} \lambda(u,f[a]) & \text{if } (u,f[a]) \in \text{dom}(\lambda), \\ \text{null} & \text{else}. \end{cases} \]

\[ [f_\alpha]_G^{\text{on}} = \begin{cases} [\phi_1]_G \cdots [\phi_k]_G & \text{if } t \in \text{OT} \text{ and } \tau(u) = t, \\ \epsilon & \text{else (}\epsilon\text{ denotes the empty word).} \end{cases} \]

\[ [\phi_1 \cdots \phi_k]_G^{\text{on}} = [\phi_1]_G^{\text{on}} \cdots [\phi_k]_G^{\text{on}}. \]

In this work, we assume that \( A \subset F \). More specifically, we restrict the set of arguments of a field \( f \in F \) to be the set of scalar fields of its type, i.e., \( \text{args}_S(f) \subseteq \{ f' | f' \in \text{fields}_S(\text{type}_S(f)), \text{type}_S(f') \in \text{Scalars} \} \). Hence, leaf fields are not assigned any arguments, and the expressions \( f_\alpha \) and \( \ell : f_\alpha \) can be written as \( f \) and \( \ell : f \), respectively [22]. In [22], the sets \( F \) and \( A \) are assumed to be disjoint; however, our assumption is in accordance with the GraphQL specification and does not affect the provided semantics.

### 2.2. Worst-case Optimal Multi-way Join Algorithms

Worst-case optimal multi-way algorithms [26] have recently gained a lot of attention (e.g., [3,27,28,29]) and have demonstrated high performance in evaluating graph pattern queries [27,29,30]. Such algorithms satisfy the AGM bound [31] and their runtime matches the worst-case size of the result of the input query [23,27]. Pair-wise join algorithms carry out join operations on two join operands at a time. Instead, worst-case optimal multi-way algorithms (e.g., Leapfrog Triejoin [32]) are recursive and evaluate input queries on a per variable basis. This evaluation method does not store any intermediate results and allows for solution mappings to be directly written to the result.

### 3. Evaluation of GraphQL Queries over RDF Graphs

In this section, we introduce the multi-way left-join algorithm that we developed for the native execution of GraphQL queries over RDF graphs. Motivated by recent results on the evaluation of basic graph pattern queries presented in [3,27], the proposed left-join algorithm is inspired by worst-case optimal multi-way join algorithms and evaluates queries on a per variable basis. However, unlike join operations, the reordering of left-join operations is not allowed. Hence, we have to pay attention to the order in which the variables of a query are evaluated. In addition, left-join operations might produce partial solutions (i.e., solutions with \text{null} values in the context of GraphQL). To respect the or-
Consider the queries of Example 2.2. The operands of Example 3.1 demonstrate the operand generation process of GraphQL queries. The operands are assigned the variable that is already assigned to the operand of their field. Consequently, vertices representing people whose last name is not "Doe" will be discarded. The inner fields are associated with the root field through the same variable. The target vertices of fields are of specific type. To restrict the type of target vertices, an additional operand is generated using the pattern of root fields presented above. In practice, this additional operand can be omitted, if the schema allows it (e.g., via a directive). Provided an expression \( f[\alpha] \{ \phi \} \), the operand of an argument-value pair \( a = (f', v) \in \alpha \) is created by \( \langle ?\text{var}, f', v \rangle \). Last, the operand of an inline fragment \( \ell : f[\alpha] \{ \phi \} \) and \( \ell : f[\alpha] \) are generated using the patterns of \( f[\alpha] \{ \phi \} \) and \( f[\alpha] \), respectively.

Two query operands participate in a (left-)join operation, if they share a variable. For assigning variables to operands, we take advantage of the hierarchical structure of GraphQL. More specifically, the target vertices of a field and the source vertices of its nested fields share the same variable. The operands of inline fragments are also assigned the variable of the target vertices of their parent fields. In the case of arguments, their nested fields, share the same variable. The operands of inline fragments are also assigned GraphQL schema to RDF terms. Our implementation computes this mapping using a GraphQL directive [25, Section 3.13]. In the following, we omit this mapping for brevity.

In a GraphQL query, we distinguish three types of fields: i) root fields, ii) inner fields, and iii) leaf fields. The root field of a query is the starting point of the traversal. Its corresponding operand should only contain the entities of the underlying RDF graph that are instances of its type. The pattern \( \langle ?\text{var}, \text{rdf:} \text{type}, \text{type}_{S}(f) \rangle \) is used to extract these instances, with \( ?\text{var} \) being a variable that will be assigned the extracted instances. Inner and leaf fields represent edges between a source and a target vertex in the graph and their operands are created using patterns of the form \( \langle ?\text{var}_1, f, ?\text{var}_2 \rangle \). Ultimately, \( ?\text{var}_1 \) will be assigned the source vertices of the edge, whereas \( ?\text{var}_2 \) will be assigned the target vertices. In the case of inner and leaf fields, we need to also consider the type of the target vertices. More specifically, in RDF, the objects of properties can vary in type, whereas, in GraphQL, the target vertices of fields are of specific type. To restrict the type of target vertices, an additional operand is generated using the pattern of root fields presented above. In practice, this additional operand can be omitted, if the schema allows it (e.g., via a directive). Provided an expression \( f[\alpha] \{ \phi \} \), the operand of an argument-value pair \( a = (f', v) \in \alpha \) is created by \( \langle ?\text{var}, f', v \rangle \). Last, the operand of an inline fragment \( \ell : f[\alpha] \{ \phi \} \) is executed only if the parent field is an instance of the type \( t \), is created by \( \langle ?\text{var}, \text{rdf:} \text{type}, t \rangle \). The operands of the aliased fields \( \ell : f[\alpha] \{ \phi \} \) and \( \ell : f[\alpha] \) are generated using the patterns of \( f[\alpha] \{ \phi \} \) and \( f[\alpha] \), respectively.

Example 3.1 Consider the queries of Example 2.2. The operands of \( \Phi_1 \) are generated by the patterns: 1) \( \langle x, \text{rdf:} \text{type}, \text{Person} \rangle \), 2) \( \langle x, \text{name}, "Doe" \rangle \), 3) \( \langle x, \text{fname}, ?y \rangle \), and 4) \( \langle x, \text{email}, ?z \rangle \). Note that the operands of the root field and its argument share the same variable. Consequently, vertices representing people whose last name is not "Doe" will be discarded. The inner fields are associated with the root field through the
variable \( ?x \). Also note that the target vertices of the inner fields are assigned different variables. The operands of \( \varphi \) are created in a similar manner and their corresponding patterns are: 1) \( \langle ?x, \text{rdf:type, Company} \rangle \), 2) \( \langle ?x, \text{name, ?y} \rangle \), 3) \( \langle ?x, \text{employees, ?z} \rangle \), 4) \( \langle ?z, \text{rdf:type, Person} \rangle \), 5) \( \langle ?z, \text{name, ?w} \rangle \), and 6) \( \langle ?z, \text{id, ?v} \rangle \).

The inner fields \text{name and employees} are associated with the root field \text{companies} through the variable \( ?x \), whereas the operands of the leaf fields \text{id} and \text{name} are associated with the operand of their parent field, namely \text{employees}, through \( ?z \). Last, note the additional operand that is generated for the field \text{employees}. Its goal is to discard vertices that are not of type \text{Person}. We assume that type filtering is not required for scalar types for brevity.

### 3.2. Operand Dependency Graph

The operand dependency graph is inspired by pattern trees [33] and captures the dependencies between the operands of a query. If an operand is not successfully resolved during the query evaluation, its dependent operands should not be evaluated. For example, provided a GraphQL query \( f[\alpha]\{\varphi\} \), the operands of \( \varphi \) should not be considered if the operands of \( f[\alpha] \) do not produce any results. However, if the operands of \( \varphi \) do not produce any results, the results generated by \( f[\alpha] \) should not be discarded. The operand dependency graph is formally defined as follows.

**Definition 3.1 (Operand dependency graph)** Let \( O \) be a list of query operands and \( \Sigma \) an alphabet. Furthermore, let \( \mathbb{N} = \{i \in \mathbb{N} \mid 1 \leq i \leq n\} \). An operand dependency graph is a directed vertex-edge-labelled graph \( G = (V, E) \), where \( V =\mathbb{N}\cup O \) and \( E \subseteq V \times \Sigma \times V \).

An operand \( v \in V \) depends on operand \( u \in V \), if and only if \( \exists e \in E \) such that \( e = (u, \sigma, v) \) and \( \sigma \in \Sigma \).

As per Definition 3.1, the vertices of an operand dependency graph correspond to the operands of its respective query. The variables appearing in query operands are assigned unique labels stemming from \( \Sigma \) and are used to label the vertices and edges of the dependency graph. The vertices of the dependency graph are assigned the labels of their respective operands’ variables. Two operands are connected via an edge only if they share a variable. The label of an edge is determined by the label shared by its incident vertices.

For the construction of the operand dependency graph, we take advantage of the hierarchical structure of GraphQL queries. Provided an expression \( f[\alpha]\{\varphi\} \) (\( \epsilon : f[\alpha]\{\varphi\} \)), the operands of \( f[\alpha] \) comprise a strongly connected component in the dependency graph, as they all depend on each other. This means that any vertex \( v \) of \( f[\alpha] \) is reachable from any other vertex \( u \) of \( f[\alpha] \), with \( v \neq u \), provided that \( f[\alpha] \) generates multiple operands. As the operands of \( \varphi \) depend on the operands of \( f[\alpha] \), the vertices of \( f[\alpha] \) are not reachable from the vertices of \( \varphi \). In the case of \( \alpha \text{ on } t\{\varphi\} \) expressions, the operands of \( \varphi \) depend on the operand of \( t \). This means that the vertex of \( t \) and the vertices of \( \varphi \) are connected with edges, whose source is the vertex corresponding to \( t \). In the case of \( \varphi_1 \ldots \varphi_k \) expressions, there are not any edges between the vertices of any \( \varphi_i \) and \( \varphi_j \), with \( 1 \leq i, j \leq k \) and \( i \neq j \), as the evaluation of \( \varphi_i \) does not affect the evaluation of \( \varphi_j \) (Definition 2.4). Operands that depend on each other participate in join operations, whereas unidirectional edges denote left-join operations. Last, in the multi-way left-join algorithm, which is presented below, we make use of the root node of the directed acyclic graph connecting the strongly connected components of an operand dependency graph. Herein, we refer to this node as the independent strong component of the dependency graph.
Example 3.2 The operand dependency graphs corresponding to the GraphQL queries Example 2.2 and their respective operands (Example 3.1) are as follows.

In the operand dependency graph of \( \phi_1 \) (left-hand side), there are not any edges connecting the vertices of operands 3 and 4, since the evaluation of 3 does not affect the evaluation of 4, and vice versa. The independent strong component of the dependency graph consists of the operands 1 and 2. In the operand dependency graph of \( \phi_2 \) (right-hand side), operands 3 and 4 depend on each other. Both operands are generated by the inner field employees, with operand 4 being responsible for removing any RDF terms assigned to \( ?z \) that are not of type Person. In this case, the independent strong component consists of a single vertex, namely the vertex corresponding to operand 1.

3.3. Multi-way Left-Join Algorithm

Here, we present our multi-way left-join algorithm (Algorithm 1) for the evaluation of GraphQL queries over RDF graphs. The key characteristics of our approach are the following. First, it evaluates join and left-join operations on a variable simultaneously. Second, it uses the operand dependency graph to eliminate the transitively dependent operands of an empty operand (i.e., an operand that is not successfully resolved), thus avoiding unnecessary operations.

The function \( \text{MWLJ} \) (lines 1–5) takes as input a GraphQL query and is responsible for generating the operands of the query (line 2) and their dependency graph (line 3). For simplicity, we assume throughout the algorithm that the operands are stored within the vertices of the graph. This function is also responsible for initializing the solution mapping, which stores the bindings of all variables of the query, as its domain is equal to the set of labels appearing in the query’s operand dependency graph. Recall that each variable is assigned a unique label (Section 3.2). After initializing the solution mapping, \( \text{MWLJ} \) calls the recursive function \( \text{MWLJREC} \) (line 5), which takes the operands dependency graph \( G \) and the solution mapping \( X \) as inputs.

The function \( \text{MWLJREC} \) (lines 6–22) is responsible for carrying out the join and left-join operations and generating the solutions of the query. In case the provided dependency graph is disconnected, \( \text{MWLJREC} \) is called for each connected component of the graph (lines 7–9). Disconnected dependency graphs correspond to \( \phi_1 \ldots \phi_k \) expressions, as there are no dependencies between any \( \phi_i \) and \( \phi_j \), with \( 1 \leq i, j \leq k \) and \( i \neq j \) (Section 3.2). If the provided graph is not strongly connected, there are left-join operations that need to be carried out (lines 10–20). To respect the order of left-join operations, the algorithm focuses on the set of labels (i.e., variables) \( U \) that are found in the independent strong component of the dependency graph (line 11). For the GraphQL queries that we consider in this work, the set \( U \) contains only a single label. This will not be the case once we take GraphQL’s input object types \([25\text{, Section 3.10}]\) into consideration.\(^3\) After

\(^3\)Note that also the formal definitions of GraphQL in \([22]\) do not consider input object types.
Algorithm 1 Multi-way Left-join Algorithm

1: function MWLJ(Q)  \text{Q: Input GraphQL query}  
2: \begin{align*}  O \leftarrow & \text{generate the operands of } Q \\  G \leftarrow & \text{create the operand dependency graph of } Q \text{ using } O \\  X \leftarrow & \text{initialize solution mapping with domain equal to the set of labels appearing in } G \\  \text{MWLJ} \text{REC}(G, X) \end{align*}  
3: function MWLJ \text{REC}(G, X)  \text{G: operand dependency graph, X: solution mapping}  
4: if G is disconnected then  \text{Evaluation of } \phi_1 \ldots \phi_k \text{ expressions}  
5: for all connected components \( G_i \) of \( G \) do  \text{Each } G_i \text{ corresponds to a } \phi_i, 1 \leq i \leq k  
6: MWLJ \text{REC}(G_i, X)  
7: else if \( G \) is not strongly connected then  \text{Left-join operation}  
8: \begin{align*}  U \leftarrow & \text{the set of labels appearing in the independent strong component of } G \\  x \leftarrow & \text{select a label from } U \\  \text{for all values } \chi \text{ of } x \do & \text{Carries out join and left-joins simultaneously} \\  G' \leftarrow & \text{prune vertices of empty operands and their transitively dependent vertices from } G \\  \text{if } G' \text{ is empty then} \\  \text{continue} & \text{All operands are pruned (unsuccessful join); continue with the next } \chi \\  \text{update the value of } x \text{ in } X \text{ with } \chi & \text{Join operations were successful} \\  \text{remove } x \text{ from } G' \text{; remove vertices without any labels from } G' \\  \text{MWLJ} \text{REC}(G', X) \end{align*}  
9: else  \text{G is strongly connected (no left-join operations)}  
10: MWJ(G, X)  \text{Carry out multi-way join (no more left joins after this point)} 

selecting a label \( x \) from \( U \), the algorithm iterates over all possible values of \( x \) and carries out all join and left-join operations on \( x \) (line 14). The algorithm proceeds by removing any operands that were not successfully resolved along with their transitively dependent operands, which can be found by traversing the dependency graph (line 15). If the resulting graph \( G' \) ends up being empty, a join operation was not successful and the algorithm continues with the next value of \( x \) (lines 16–17). If \( G' \) is not empty, the solution mapping \( X \) is updated with the current value of \( x \), which is removed from \( G' \) along with any fully resolved operands, and the algorithm proceeds with the next recursive step (lines 18–20). In case the provided graph \( G \) is strongly connected, the algorithm proceeds with a multi-way join algorithm, as there are no left-join operations left to be carried out. The active solution mapping \( X \) will be ultimately projected once the remaining join operations are carried out by the multi-way join algorithm.

Example 3.3 Consider the query \( \phi_1 \) of Example 2.2. Provided the example RDF graph

\begin{verbatim}
<p1> rdf:type <Person>; <lname> "Doe"; <fname> "Jon"; <email> "e1".
<p2> rdf:type <Person>; <lname> "Doe"; <fname> "Jan".
the proposed algorithm will produce three solutions: \{x:p1, y:"Jon"\}, \{x:p1, z:"e1"\}, and \{x:p2, y:"Jan"\}. The algorithm selects first the label corresponding to the variable \( x \), which is assigned the identifiers of people in the graph. For the value \( p1 \) of \( x \), the algorithm generates two solutions. The first one provides the first name (fname) of \( p1 \), which is assigned to \( y \), whereas the second one provides its email, which is assigned to \( z \). For the value \( p2 \), the algorithm generates only one solution, as \( p2 \) does not have an email in the example graph. Note that after selecting \( x \) and removing it from the operand dependency graph, the resulting dependency graph is disconnected. Variables that do not appear in a solution mapping are unbound in that particular mapping.
\end{verbatim}
Regarding the enumeration of GraphQL queries, in [22], the authors study the enumeration problem for GraphQL queries that are non-redundant and in ground-typed normal form. Recall that such queries can be computed in time linear to the size of their response (Section 2.1). Each solution mapping generated by our algorithm captures a unique path of the response corresponding to the provided query. As our left-join algorithm computes a solution mapping entirely, we are able to directly construct the path that corresponds to a particular solution mapping, once it is evaluated. In addition, due to the recursive nature of our algorithm, the solution mappings of the sub-trees of a particular node of a GraphQL response share common values (Example 3.3). Hence, we are able to avoid visiting the nodes of a response multiple times.

3.4. Implementation

We have implemented the proposed algorithm within the tensor-based triple store Tentris [3]. Tentris achieves state-of-the-art performance in the evaluation of basic graph patterns, which are evaluated by a worst-case optimal multi-way join algorithm [3,12]. Our implementation, namely TentrisGQL, uses Tentris’ multi-way join algorithm (Algorithm 1, line 22), and tensor slicing operations to generate the operands of GraphQL queries.

To bridge the gap between GraphQL schemata and RDF graphs, we follow Neo4j’s example and define several directives in our implementation. As per the GraphQL specification, “directives can be used to describe additional information for types, fields, fragments and operations” [25, Section 3.13]. We mentioned in Section 3.1 that GraphQL types and fields need to be mapped to RDF terms. To this end, we define in our implementation the directive @uri. For example, the type definition type Person @uri(value: “http://www.exmpl.org/Person”) maps the type Person to the RDF term http://www.exmpl.org/Person. As the inverse of a property is not always available in RDF graphs, we also define the field directive @inverse, which denotes that the inverse direction of a field’s property should be used. Last, we introduce the field directive @filter, which denotes that the results of a particular field should be filtered using that field’s type. This directive should be used on fields that are mapped to properties having ranges consisting of multiple RDF classes (Section 3.1).

4. Experimental Results

In this section, we present the performance evaluation of TentrisGQL, which we evaluated using the Linköping GraphQL Benchmark (LinkGBM) [24]. LinkGBM is a synthetic benchmark generator that provides a GraphQL schema that captures the structure of the generated datasets, and a set of 16 GraphQL query templates. To the best of our knowledge, LinkGBM is currently the only publicly available benchmark for evaluating GraphQL services. The experiments that are presented below were carried out on a Debian 10 server with an AMD EPYC 7742 64-Core Processor, 1TB RAM, and two 3TB NVMe SSDs in RAID 0. All artifacts (e.g., datasets, GraphQL schemata, queries, and system configurations) are available online.\(^4\)

\(^4\)https://github.com/neon-graphQL/neon-graphql-js/blob/master/docs/graphql-schema-directives.md

\(^5\)https://github.com/dice-group/graphql-benchmark
4.1. Systems

As baseline for our experiments, we used Neo4j Community Edition 5.5.0 \cite{11}. We selected Neo4j because it is a widely used graph database and it provides its own tools for processing GraphQL queries. In our experiments, we evaluated Neo4j in two different modes. In the first mode (Neo4jC), Neo4j was provided with Cypher queries instead of GraphQL queries. The GraphQL queries used in our experiments (Section 4.2) were translated to Cypher queries using a library provided by Neo4j\cite{6}. The purpose of this mode was to compare the query evaluation performance of TentrisGQL against that of Neo4j, as no result rewriting takes place in this mode of Neo4j. To find out the overhead introduced by the process of result rewriting, we used a second mode, namely Neo4jGQL. Neo4jGQL includes an external application that is connected to Neo4j and is responsible for translating GraphQL queries to Cypher queries and rewriting query results to GraphQL responses.\cite{7} Recall that TentrisGQL incrementally constructs GraphQL responses. For the evaluation, we used Neo4j’s recommended memory settings\cite{8} and built the appropriate search indices. More specifically, regarding the memory settings, we allocated 31GB of memory to the Java virtual machine (JVM) and 957GB for caching purposes. In our experiments, we also evaluated TentrisGQLBase, a version of TentrisGQL that treats fields of type ID (i.e., fields that capture IRIs of RDF terms) as strings. As a result, TentrisGQLBase needs to carry out left joins and joins to evaluate such fields when they appear as leaf fields or arguments in a query, respectively. In contrast, TentrisGQL accesses the IRIs of RDF terms directly. TentrisGQLBase provides us with insights on the impact that the evaluation of leaf fields has on the performance of our service.

4.2. Datasets, Query Templates, and Schema

LinGBM’s dataset generator relies on the dataset generator of LUBM \cite{35} and allows for the generation of datasets of varying sizes via the use of a scale factor. To evaluate the performance of our approach on RDF graphs of different sizes, we generated three graphs (Table 1), namely LinGBM100, LinGBM500, and LinGBM1000. For our experiments, we modified LinGBM’s dataset generator to include the classes corresponding to the interface types of the schema, as both systems expect them to be stated in the input data.

As previously mentioned, LinGBM also provides a set of query templates. Their design follows a choke-point methodology \cite[Section 3.3]{24}; each choke-point focuses on a particular workload or operation. In this work, we are interested in join and left-join operations. Hence, we focus on the choke-points Attribute Retrieval (CP1) and Relationship Traversal (CP2) of LinGBM. There are only six query templates (QT1-QT6) that focus exclusively on CP1 and CP2. To include additional queries in our evaluation, we modified the query templates QT7-QT14 by removing those features that are not related to CP1 and CP2 (e.g., ordering, filtering, and pagination). In addition, we had to remove input objects from the query templates QT11-QT14, as they are currently not supported by our implementation. Ultimately, in our experiments, we used 11 query templates and two non-parameterized queries (Table 2).\cite{9}

\footnotesize{\textsuperscript{6}}https://github.com/neo4j-graphql/neo4j-graphql-js  
\footnotesize{\textsuperscript{7}}We followed the example used in https://github.com/neo4j-graphql/neo4j-graphql-js.  
\footnotesize{\textsuperscript{8}}https://neo4j.com/docs/operations-manual/5/tools/ne04j-admin/ne04j-admin-nemrec  
\footnotesize{\textsuperscript{9}}After our modifications, QT8 and QT11 do not have any parameters, and QT13 and QT14 are identical.
Table 1. The datasets used in the experiments.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>#Triples</th>
<th>#Distinct Subjects</th>
<th>#Distinct Predicates</th>
<th>#Distinct Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinGBM100</td>
<td>100</td>
<td>16M</td>
<td>2M</td>
<td>20</td>
</tr>
<tr>
<td>LinGBM500</td>
<td>500</td>
<td>79M</td>
<td>10M</td>
<td>20</td>
</tr>
<tr>
<td>LinGBM1000</td>
<td>1000</td>
<td>160M</td>
<td>21M</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. The depth (D) and the average size of the GraphQL response (aRS-SF) in bytes of each GraphQL query template (QT) for each scale factor (SF). QT8 and QT11 are not parameterized.

<table>
<thead>
<tr>
<th>QT</th>
<th>D</th>
<th>aRS-100</th>
<th>aRS-500</th>
<th>aRS-1000</th>
<th>QT</th>
<th>D</th>
<th>aRS-100</th>
<th>aRS-500</th>
<th>aRS-1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>34K</td>
<td>170K</td>
<td>338K</td>
<td>8</td>
<td>1</td>
<td>4M</td>
<td>20M</td>
<td>40M</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>21K</td>
<td>105K</td>
<td>200K</td>
<td>9</td>
<td>4</td>
<td>279K</td>
<td>1.3M</td>
<td>2.7M</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>10</td>
<td>1</td>
<td>6.3M</td>
<td>31M</td>
<td>63M</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>81K</td>
<td>425K</td>
<td>864K</td>
<td>11</td>
<td>2</td>
<td>3.7M</td>
<td>18M</td>
<td>37M</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>12M</td>
<td>311M</td>
<td>1.2G</td>
<td>12</td>
<td>3</td>
<td>79K</td>
<td>397K</td>
<td>785K</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>19K</td>
<td>94K</td>
<td>192K</td>
<td>13</td>
<td>3</td>
<td>73K</td>
<td>373K</td>
<td>738K</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>20K</td>
<td>101K</td>
<td>202K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The GraphQL schema provided by LinGBM is meant to be used by GraphQL services that do not generate GraphQL schemata automatically (e.g., relational schema to GraphQL schema). For our experiments, we modified the provided schema by removing those types and features that are not required by the query templates (e.g., input types and enumeration types) used in the experiments. For TentrisGQL, we extended the schema with the directives of our implementation (Section 3.4). In a similar manner, we extended the schema used for Neo4j with Neo4j’s respective directives.

4.3. Benchmark Configurations and Execution

Our GraphQL service was evaluated on two different benchmark configurations. The purpose of the first configuration was to evaluate the performance of our service on each query template. For each template, we created a stress test consisting of ten query instances per template (110 queries). We also created a stress test for each non-parameterized query (112 queries in total). The stress tests were executed 5 consecutive times and independently from each other, thus ensuring that the query instances were executed the same number of times. With the second configuration, we measured the performance of our system when queried by multiple clients. This configuration consisted of one stress test, which included one query instance from each template and the two non-parameterized queries (i.e., 13 queries). During the execution of the second configuration, it was important that there were multiple clients issuing queries at all times. For this reason, we configured the clients to issue queries concurrently for one hour [3, 36]. In both configurations, the execution of each stress test is preceded by a warm-up run, in which the queries of the corresponding stress test are executed once. This allowed Neo4j to load and cache its data structures in the main memory.

The stress tests of both benchmark configurations were executed over HTTP using the benchmark execution framework IGUANA in version 3.3.0 [36]. As in [3], we set
the timeout across all benchmarks to three minutes and measured the performance of our implementation using the number of queries executed per second (QPS) and the penalized average QPS (pAvgQPS); the penalty for failed queries (e.g., timed out queries) was set to three minutes. Last, we compared the results generated by all systems to ensure that they return the same results across all queries.

<table>
<thead>
<tr>
<th></th>
<th>Neo4jC</th>
<th>Neo4jGQL</th>
<th>TentrisGQLBase</th>
<th>TentrisGQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinGBM100</td>
<td>230.32</td>
<td>91.51</td>
<td>366.27</td>
<td>524.90</td>
</tr>
<tr>
<td>LinGBM500</td>
<td>72.13</td>
<td>33.72</td>
<td>164.23</td>
<td>229.90</td>
</tr>
<tr>
<td>LinGBM1000</td>
<td>43.99</td>
<td>20.11</td>
<td>124.59</td>
<td>169.79</td>
</tr>
</tbody>
</table>

Table 3. Overhead of result rewriting in Neo4jGQL (pAvgQPS).
4.4. Results

The results of the first benchmark configuration are presented in Figures 1 and 2. Figure 1 shows that both TentrisGQL and TentrisGQLBase outperform Neo4jC across all query templates in all datasets, with TentrisGQL achieving 1.5 (QT2 and QT13) to 7.4 (QT3) times higher pAvgQPS than Neo4jC in the largest dataset, namely LinGBM1000. In addition, both TentrisGQL and TentrisGQLBase achieve higher median QPS than Neo4jC in all datasets (Figure 2). Figure 3 summarizes the results reported in the second benchmark configuration. We removed the query instance corresponding to QT5 from the second configuration’s query list because Neo4j was running out of JVM memory when this query was issued by multiple clients. TentrisGQL and TentrisGQLBase did not face any memory-related issues. Figure 3 shows that Neo4jC scales better than TentrisGQL and TentrisGQLBase when queried by 4 and 8 concurrent clients in the smallest dataset (i.e., LinGBM1000). However, TentrisGQL and TentrisGQLBase achieve higher pAvgQPS than Neo4jC in all cases and in particular, TentrisGQL achieves 3.5 higher pAvgQPS in the case of 16 clients in LinGBM1000. To measure the overhead introduced by the rewriting of Neo4j’s results to GraphQL responses, we used the second benchmark configuration with one concurrent user. Table 3 shows that rewriting process leads to TentrisGQL achieving up to 8.4 times higher pAvgQPS than Neo4jGQL.

4.5. Discussion

The performance of the systems did not vary significantly across all datasets. In particular, they were not significantly affected by the increasing size of the datasets. The systems’ performance was mostly affected by the average result size (aRS) of the query templates (Table 2). In particular, all systems achieved their highest and lowest pAvgQPS in all datasets in QT3 and QT5, respectively. QT3 has the lowest aRS, whereas QT5 has the highest. The depth of the query templates also affects the systems’ performance. For example, the pAvgQPS of the systems in QT6 is lower than in QT7, even though the latter has a higher aRS. Queries with higher values of depth require more left-join operations in TentrisGQL and longer path traversals in Neo4j.

Another factor that impacts the performance of our GraphQL service is the size of the operands corresponding to leaf fields, which are evaluated via left-join operations. This observation is grounded in the performance of TentrisGQLBase (Figure 1), which is always equivalent to, or worse than, TentrisGQL’s performance. Recall that TentrisGQLBase, unlike TentrisGQL, evaluates leaf fields and arguments of type ID via left-join and join operations, respectively. Neo4j employs the property graph model, which allows it to represent leaf fields as node properties. Hence, for evaluating leaf fields, Neo4j does not iterate over all of a particular property key’s properties. Despite these additional operations, both TentrisGQL and TentrisGQLBase outperform Neo4j. This suggests that our algorithm does not introduce much overhead to the computations.

The results of Table 3 are in line with the results reported in [18] and demonstrate the importance of GraphQL services being able to directly construct GraphQL responses. Regarding the memory usage, we measured the memory used by the systems in LinGBM1000 when queried by 16 concurrent clients using `pmap`\(^\text{10}\). The highest Resident Set Size (RSS) reported by TentrisGQL and Neo4jC was 41GB and 44GB, respectively.

\(^{10}\text{https://linux.die.net/man/1/pmap}\)
5. Related Work

Recently, several graph storage solutions have made efforts to allow users to access their data via GraphQL. Dgraph [16] is a distributed graph database that natively supports GraphQL. It also provides its own query language, namely DQL. In Dgraph, GraphQL operations are translated to DQL operations. However, response objects are constructed following the GraphQL specification. Hence, a rewriting of the results is not required. We did not include Dgraph in our experiments for two reasons. First, Dgraph does not fully support RDF, as it is not able to handle URIs. Additionally, Dgraph’s GraphQL service expects predicates to be prefixed with their subject’s type. Consequently, existing RDF graphs need to be substantially modified to be stored in a Dgraph instance. Second, Dgraph does not provide a bulk loader for its GraphQL service; hence it is not able to load large knowledge graphs efficiently. In addition to the translation tools used in Section 4, Neo4j provides a library that serves as a middleware between applications and database instances. This library is responsible for the translation process of GraphQL queries to Cypher queries. Regarding triple stores, Stardog [37] and the commercial edition of GraphDB [9] provide GraphQL support by translating GraphQL to SPARQL. Virtuoso [8] introduced a GraphQL plugin that allows its users to query RDF graphs via GraphQL. To bridge the gap between GraphQL and SPARQL, this plugin relies on OWL ontologies to map the types and fields of GraphQL schemata to RDF terms. We did not include Virtuoso in our experiments as it does not perform type filtering in inner fields, which leads to queries returning incorrect results.

6. Conclusion and Future Work

We presented an approach for the native evaluation of GraphQL queries over RDF graphs. As GraphQL queries require left-join operations, we focused on the development of a novel multi-way left-join algorithm that is inspired by worst-case optimal multi-way join algorithms. Similarly to worst-case optimal multi-way join algorithms, the proposed left-join algorithm recursively evaluates queries on a per variable basis, which allows for the incremental enumeration of GraphQL queries. By implementing our approach within the tensor-based triple store Tentris, we provide the first publicly available triple store that treats GraphQL as a first-class citizen. The performance evaluation of our implementation demonstrates the efficiency of the left-join algorithm, as our implementation outperforms a state-of-the-art graph database, namely Neo4j.

Our implementation currently supports the features of the language that are required by its formal semantics (Equation 1). Our future work will focus on extending our GraphQL service with all features from the specification. To the best of our knowledge, there have not been any works that focus on evaluating SPARQL queries requiring left-join operations on a variable basis. To this end, we plan to use our approach for the evaluation of such SPARQL queries (i.e., queries containing optional graph patterns).

11https://discuss.dgraph.io/t/graphql-vs-dql-dgraph-blog/14311; see paragraph “When not to use GraphQL”.
12https://neo4j.com/docs/graphql-manual/current/
13https://community.openlinksw.com/t/introducing-native-graphql-support-in-virtuoso/3378
14https://github.com/openlink/virtuoso-opensource/issues/1115
References


