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-

cently, 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 *Fields* be an infinite set of field names and $F \subset$ *Fields* a finite subset of *Fields*. Let A and T be finite sets of argument names and type names, respectively, where T is the disjoint union of O_T (object type names), I_T (interface type names), U_T (union type names) and *Scalars* (scalar type names). Last, let $L_T = \{[t] \mid t \in T\}$ be the set of list types constructed from T and *Vals* be a set of scalar values.

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

- *$fields_{\mathcal{S}} : (O_T \cup I_T) \to 2^F$ that assigns a set of fields to every object type and every interface type,*
- *$args_{\mathcal{S}} : F \to 2^A$ that assigns a set of arguments to every field,*
- *$type_{\mathcal{S}} : F \cup A \to T \cup L_T$ that assigns a type or a list type to every field and argument, where arguments are assigned scalar types,*
- *$union_{\mathcal{S}} : U_T \to 2^{O_T}$ that assigns a nonempty set of object types to every union type,*
- *$impl_{\mathcal{S}} : I_T \to 2^{O_T}$ that assigns a set of object types to every interface,*
- *$root_{\mathcal{S}} \in O_T$ that represents the* query *root type.*

Example 2.1 *Consider the following GraphQL schema $\mathcal{S}$*

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

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

$args_{\mathcal{S}}($people$) = \{$lname$\}$, $fields_{\mathcal{S}}($Entity$) = \{$id, email$\}$, $type_{\mathcal{S}}($id$) = $String, $fields_{\mathcal{S}}($Person$) = \{$id, fname, lname, email, age$\}$, $root_{\mathcal{S}} = $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 Vals,*
- *$\tau : N \to O_T$ is a function that assigns a type to every node,*
- *λ is a partial function that assigns a scalar value $v \in Vals$ or a sequence $[v_1 \dots v_n]$ of scalar values ($v_i \in Vals$) 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 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 ϕ constructed from the following grammar where* `[,]`, `{`, `}`, `:` *and* `on` *are terminal symbols, $t \in O_T \cup I_T \cup U_T$, $f \in F$, $\ell \in Fields$, and α is a partial mapping from A to Vals:*

$$\phi ::= f\,[\alpha]\mid \ell : f\,[\alpha]\mid on\,t\{\phi\}\mid f\,[\alpha]\{\phi\}\mid \ell : f\,[\alpha]\{\phi\}\mid \phi \dots \phi\,.$$

Example 2.2 *Examples of GraphQL queries conforming to the GraphQL schema $\mathcal{S}$ of Example 2.1 are the following:*

$$\phi_1 = \texttt{people(lname: "Doe") \{ fname email \}} \quad and$$
$$\phi_2 = \texttt{companies \{ name employees \{ id lname \} \}}\,.$$

Both queries demonstrate the hierarchical structure of GraphQL queries. For example, ϕ_2 accesses fields in the first level that belong to the object type `Company`*. In the second level, it accesses fields of the object type* `Person`*, as* $type_\mathcal{S}(\texttt{employees}) = \texttt{[Person]}$*.*

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 ϕ a non-redundant GraphQL query in ground-typed normal form, both conforming to a GraphQL schema $\mathcal{S}$ over (F,A,T). The evaluation of ϕ over G from node $u \in N$, denoted by $[\![\phi]\!]_G^u$, is captured by Equation 1.[2] The evaluation of ϕ over G, denoted by $[\![\phi]\!]_G$, is simply $[\![\phi]\!]_G^\rho$.*

---

[2]The expressions $\ell : f\,[\alpha]\{\phi\}$ and $\ell : f\,[\alpha]$ are evaluated by replacing f with ℓ in the first two rules' results.

$$\llbracket f[\alpha] \rrbracket_G^u = \begin{cases} f : \lambda(u, f[a]) & \text{if } (u, f[a]) \in dom(\lambda), \\ f : null & \text{else.} \end{cases}$$

$$\llbracket f[\alpha]\{\phi\} \rrbracket_G^u = \begin{cases} f : [\{\llbracket \phi \rrbracket_G^{v_1}\} \dots \llbracket \phi \rrbracket_G^{v_k}] & \begin{aligned}&\text{if } type_{\mathcal{S}}(f) \in L_T \text{ and} \\ &\{v_1 \dots v_k\} = \{v_i \mid (u, f[a], v_i) \in E\},\end{aligned} \\[2ex] f : \{\llbracket \phi \rrbracket_G^v\} & \begin{aligned}&\text{if } type_{\mathcal{S}}(f) \notin L_T \text{ and} \\ &(u, f[a], v) \in E,\end{aligned} \\[2ex] f : null & \begin{aligned}&\text{if } type_{\mathcal{S}}(f) \notin L_T \text{ and there is no} \\ &v \in N \text{ s.t. } (u, f[a], v) \in E.\end{aligned} \end{cases} \qquad (1)$$

$$\llbracket \text{on } t\{\phi\} \rrbracket_G^u = \begin{cases} \llbracket \phi \rrbracket_G^u & \text{if } t \in O_T \text{ and } \tau(u) = t, \\ \varepsilon & \text{else } (\varepsilon \text{ denotes the empty word).} \end{cases}$$

$$\llbracket \phi_1 \dots \phi_k \rrbracket_G^u = \llbracket \phi_1 \rrbracket_G^u \dots \llbracket \phi_k \rrbracket_G^u.$$

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., $args_{\mathcal{S}}(f) \subseteq \{f' \mid f' \in fields_{\mathcal{S}}(type_{\mathcal{S}}(f)), type_{\mathcal{S}}(f') \in Scalars\}$. 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 algorithm 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 null values in the context of GraphQL). To respect the or-

der of operations during the evaluation of a query and to ensure that partial solutions will not be discarded, we additionally introduce the *operand dependency graph*. Before introducing the operand dependency graph and the proposed multi-way left-join algorithm, we define first the process of generating the query operands of GraphQL queries.

3.1. GraphQL Query Operands

In the case of SPARQL, there are multiple features of the language that generate query operands, with the most common being the triple pattern. In the case of GraphQL, an operand needs to be generated for each *field*, *argument* and *inline fragment* of a query. Here, for simplicity, we use a notation that resembles SPARQL's triple patterns and present how to generate the operands of GraphQL queries. Note that we do not actually translate GraphQL queries to SPARQL queries. Potential implementations are free to use any means available (e.g., indices) for generating these operands. For the generation of GraphQL query operands, we must also map the types and fields of the provided 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 $\langle ?var, \texttt{rdf:type}, type_S(f) \rangle$ is used to extract these instances, with $?var$ 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 $\langle ?var_1, f, ?var_2 \rangle$. Ultimately, $?var_1$ will be assigned the source vertices of the edge, whereas $?var_2$ 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 $\texttt{a} = (f', v) \in \alpha$ is created by $\langle ?var, f', v \rangle$. Last, the operand of an inline fragment on $t\{\phi\}$, whose sub-expression ϕ is executed only if the parent field is an instance of the type t, is created by $\langle ?var, \texttt{rdf:type}, t \rangle$. 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.

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 operands are assigned the variable that is already assigned to the operand of their field. Example 3.1 demonstrates the operand generation process of GraphQL queries.

Example 3.1 *Consider the queries of Example 2.2. The operands of ϕ_1 are generated by the patterns: 1) $\langle ?x, \texttt{rdf:type}, \texttt{Person} \rangle$, 2) $\langle ?x, \texttt{lname}, \texttt{"Doe"} \rangle$, 3) $\langle ?x, \texttt{fname}, ?y \rangle$, and 4) $\langle ?x, \texttt{email}, ?z \rangle$. 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 ϕ_2 are created in a similar manner and their corresponding patterns are: 1) $\langle ?x, \texttt{rdf:type}, \texttt{Company} \rangle$, 2) $\langle ?x, \texttt{name}, ?y \rangle$, 3) $\langle ?x, \texttt{employees}, ?z \rangle$, 4) $\langle ?z, \texttt{rdf:type}, \texttt{Person} \rangle$, 5) $\langle ?z, \texttt{lname}, ?w \rangle$, and 6) $\langle ?z, \texttt{id}, ?v \rangle$. The inner fields `name` *and* `employees` *are associated with the root field* `companies` *through the variable ?x, whereas the operands of the leaf fields* `id` *and* `lname` *are associated with the operand of their parent field, namely* `employees`, *through ?z. Last, note the additional operand that is generated for the field* `employees`. *Its goal is to discard vertices that are not of type* `Person`. *We assume that type filtering is not required for scalar types for brevity.*
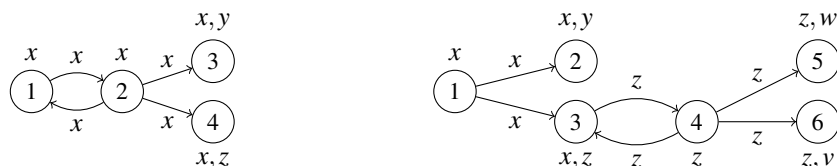
3.2. Operand Dependency Graph

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 $f[\alpha]\{\phi\}$, the operands of ϕ should not be considered if the operands of $f[\alpha]$ do not produce any results. However, if the operands of ϕ 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 Σ an alphabet. Furthermore, let $\mathbb{I}_n = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$. An operand dependency graph is a directed vertex-edge-labelled graph $G = (V, E)$, where $V = \mathbb{I}_{|O|}$ 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 Σ 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 $f[\alpha]\{\phi\}$ $(\ell : f[\alpha]\{\phi\})$, the operands of $f[\alpha]$ comprise a strong 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 ϕ depend on the operands of $f[\alpha]$, the vertices of $f[\alpha]$ are not reachable from the vertices of ϕ. In the case of $\texttt{on } t\{\phi\}$ expressions, the operands of ϕ depend on the operand of t. This means that the vertex of t and the vertices of ϕ are connected with edges, whose source is the vertex corresponding to t. In the case of $\phi_1 \ldots \phi_k$ expressions, there are not any edges between the vertices of any ϕ_i and ϕ_j, with $1 \leq i, j \leq k$ and $i \neq j$, as the evaluation of ϕ_i does not affect the evaluation of ϕ_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 ϕ_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 ϕ_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 `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, `MWLJ` calls the recursive function `MWLJ_REC` (line 5), which takes the operands dependency graph G and the solution mapping X as inputs.

The function `MWLJ_REC` (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, `MWLJ_REC` 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 ϕ_i and ϕ_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, 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)   ▷ Q: Input GraphQL query
2:     $O \leftarrow$ generate the operands of Q
3:     $G \leftarrow$ create the operand dependency graph of Q using O
4:     $X \leftarrow$ initialize solution mapping with domain equal to the set of labels appearing in G
5:     MWLJ_REC(G, X)

6: **function** MWLJ_REC(G, X)   ▷ G: operand dependency graph, X: solution mapping
7:     **if** G is disconnected **then**   ▷ Evaluation of $\phi_1 \ldots \phi_k$ expressions
8:         **for all** connected components G_i of G **do**   ▷ Each G_i corresponds to a ϕ_i, $1 \leq i \leq k$
9:             MWLJ_REC(G_i, X)
10:    **else if** G is not *strongly connected* **then**   ▷ Left-join operation
11:        $U \leftarrow$ the set of labels appearing in the independent strong component of G
12:        $x \leftarrow$ select a label from U
13:        **for all** values χ of x **do**
14:            resolve x in all operands using χ ▷ Carries out join and left joins simultaneously
15:            $G' \leftarrow$ prune vertices of empty operands and their transitively dependent vertices from G
16:            **if** G' is empty **then**
17:                **continue**   ▷ All operands are pruned (unsuccessful join); continue with the next χ
18:            update the value of x in X with χ ▷ Join operations were successful
19:            remove x from G'; remove vertices without any labels from G'
20:            MWLJ_REC(G', X)
21:    **else**   ▷ G is strongly connected (no left-join operations)
22:        MWJ(G, X)   ▷ 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 ϕ_1 of Example 2.2. Provided the example RDF graph*

```
<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.

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 [3,34]. 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[4] 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 (LinGBM) [24]. LinGBM 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, LinGBM 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 3 TB NVMe SSDs in RAID 0. All artifacts (e.g., datasets, GraphQL schemata, queries, and system configurations) are available online.[5]

---

[4] `https://github.com/neo4j-graphql/neo4j-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 [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[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.[7] Recall that TentrisGQL incrementally constructs GraphQL responses. For the evaluation, we used Neo4j's recommended memory settings[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 [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 [24, Section 3.3]; 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).[9]

---

[6]`https://github.com/neo4j-graphql/neo4j-graphql-js`

[7]We followed the example used in `https://github.com/neo4j-graphql/neo4j-graphql-js`.

[8]`https://neo4j.com/docs/operations-manual/5/tools/neo4j-admin/neo4j-admin-memrec`

[9]After our modifications, QT8 and QT11 do not have any parameters, and QT13 and QT14 are identical.

	Scale Factor	#Triples	#Distinct Subjects	#Distinct Predicates	#Distinct Objects
LinGBM100	100	16M	2M	20	3M
LinGBM500	500	79M	10M	20	18M
LinGBM1000	1000	160M	21M	20	37M

Table 1. The datasets used in the experiments.

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

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.

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
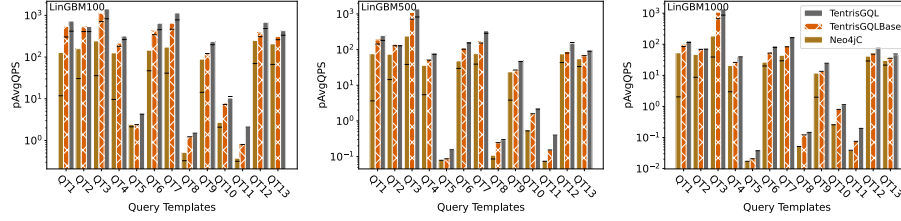
Figure 1. Performance of the systems in the first configuration w.r.t. their pAvgQPS. The black lines denote the values reported in the warmup run.
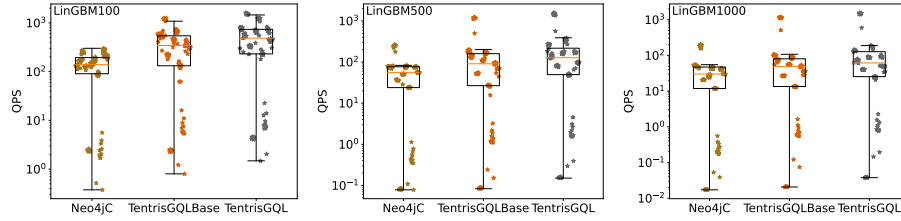

Figure 2. Performance of the systems in the first configuration w.r.t. their QPS.
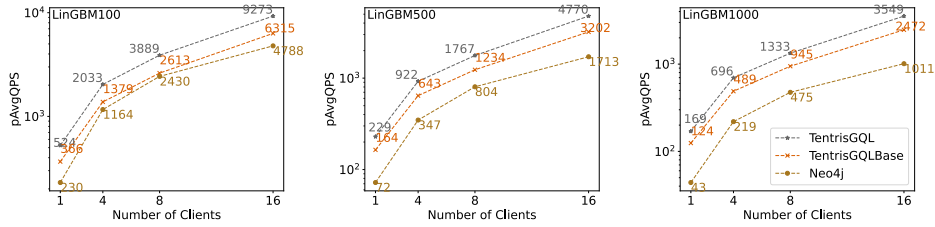

Figure 3. Scalability of the systems in the second configuration w.r.t. their pAvgQPS.

	Neo4jC	Neo4jGQL	TentrisGQLBase	TentrisGQL
LinGBM100	230.32	91.51	366.27	524.90
LinGBM500	72.13	33.72	164.23	229.90
LinGBM1000	43.99	20.11	124.59	169.79

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

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.

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., LinGBM100). 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`[10]. The highest Resident Set Size (RSS) reported by TentrisGQL and Neo4jC was 41GB and 44GB, respectively.

---

[10]`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.[11] 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[12] 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[13] 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.[14]

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).

---

[11]`https://discuss.dgraph.io/t/graphql-vs-dql-dgraph-blog/14311`; see paragraph "When not to use GraphQL".

[12]`https://neo4j.com/docs/graphql-manual/current/`

[13]`https://community.openlinksw.com/t/introducing-native-graphql-support-in-virtuoso/3378`

[14]`https://github.com/openlink/virtuoso-opensource/issues/1115`

References

[1]   Weikum G, Dong XL, Razniewski S, Suchanek FM. Machine Knowledge: Creation and Curation of Comprehensive Knowledge Bases. Found Trends Databases. 2021:108-490.

[2]   Hogan A, Blomqvist E, Cochez M, d'Amato C, de Melo G, Gutiérrez C, et al. Knowledge Graphs. ACM Comput Surv. 2021:71:1-71:37.

[3]   Bigerl A, Conrads F, Behning C, Sherif MA, Saleem M, Ngomo AN. Tentris - A Tensor-Based Triple Store. In: The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020, Proceedings, Part I; 2020. p. 56-73.

[4]   Zou L, Özsu MT, Chen L, Shen X, Huang R, Zhao D. gStore: a graph-based SPARQL query engine. VLDB J. 2014:565-90.

[5]   Yuan P, Liu P, Wu B, Jin H, Zhang W, Liu L. TripleBit: a Fast and Compact System for Large Scale RDF Data. Proc VLDB Endow. 2013:517-28.

[6]   Neumann T, Weikum G. RDF-3X: a RISC-style engine for RDF. Proc VLDB Endow. 2008;1(1):647-59.

[7]   Systap LLC. Blazegraph; N.D. Retrieved 2023-05-03. `https://blazegraph.com/`.

[8]   Openlink. Virtuoso; N.D. Retrieved 2023-05-03. `http://vos.openlinksw.com/owiki/wiki/VOS`.

[9]   Ontotext. GraphDB; N.D. Retrieved 2023-05-03. `https://www.ontotext.com/products/graphdb/`.

[10] Apache Software Foundation. Apache Jena Fuseki; N.D. Retrieved 2023-05-03. `https://jena.apache.org/documentation/fuseki2/`.

[11] Inc N. Neo4j; N.D. Retrieved 2023-03-21. `https://neo4j.com/download-center/#community`.

[12] The Linux Foundation. JanusGraph; N.D. Retrieved 2023-05-03. `https://janusgraph.org/`.

[13] Bonifati A, Fletcher GHL, Voigt H, Yakovets N. Querying Graphs. Synthesis Lectures on Data Management. Morgan & Claypool Publishers; 2018.

[14] Francis N, Green A, Guagliardo P, Libkin L, Lindaaker T, Marsault V, et al. Cypher: An Evolving Query Language for Property Graphs. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018; 2018. p. 1433-45.

[15] Rodriguez MA. The Gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015. Pittsburgh, PA, USA; 2015. p. 1-10.

[16] io D. Dgraph; N.D. Retrieved 2023-05-02. `https://dgraph.io/docs`. Available from: `https://dgraph.io/docs`.

[17] Hartig O, Hidders J. Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language. In: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Amsterdam, The Netherlands, 30 June 2019; 2019. p. 6:1-6:11.

[18] Gleim LC, Holzheim T, Koren I, Decker S. Automatic Bootstrapping of GraphQL Endpoints for RDF Triple Stores. In: Joint Proceedings of Workshops AI4LEGAL2020, NLIWOD, PROFILES 2020, QuWeDa 2020 and SEMIFORM2020 Colocated with the 19th International Semantic Web Conference (ISWC 2020), Virtual Conference, November, 2020; 2020. p. 119-34.

[19] Chaves-Fraga D, Priyatna F, Alobaid A, Corcho Ó. Exploiting Declarative Mapping Rules for Generating GraphQL Servers with Morph-GraphQL. Int J Softw Eng Knowl Eng. 2020:785-803.

[20] Taelman R, Sande MV, Verborgh R. GraphQL-LD: Linked Data Querying with GraphQL. In: Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018; 2018. .

[21] Semantic Integration LTD. HyperGraphQL; N.D. Retrieved 2023-05-03. `https://github.com/hypergraphql/hypergraphql`.

[22] Hartig O, Pérez J. Semantics and Complexity of GraphQL. In: Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018; 2018. p. 1155-64.

[23] Ngo HQ. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018; 2018. p. 111-24.

[24] Cheng S, Hartig O. LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers. In: Chbeir R, Huang ZH, Silvestri F, Manolopoulos Y, Zhang Y, editors. Web Information Systems Engineering - WISE 2022 - 23rd International Conference, Biarritz, France, November 1-3, 2022, Proceedings. vol. 13724 of Lecture Notes in Computer Science. Springer; 2022. p. 209-24.

[25]   GraphQL Specification; N.D. Retrieved 2023-03-13. `https://spec.graphql.org/June2018`.

[26]   Ngo HQ, Porat E, Ré C, Rudra A. Worst-case Optimal Join Algorithms. J ACM. 2018:16:1-16:40.

[27]   Hogan A, Riveros C, Rojas C, Soto A. A Worst-Case Optimal Join Algorithm for SPARQL. In: The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I. Springer; 2019. p. 258-75.

[28]   Arroyuelo D, Hogan A, Navarro G, Reutter JL, Rojas-Ledesma J, Soto A. Worst-Case Optimal Graph Joins in Almost No Space. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021; 2021. p. 102-14.

[29]   Freitag MJ, Bandle M, Schmidt T, Kemper A, Neumann T. Adopting Worst-Case Optimal Joins in Relational Database Systems. Proc VLDB Endow. 2020:1891-904.

[30]   Kalinsky O, Hogan A, Mishali O, Etsion Y, Kimelfeld B. Exploration of Knowledge Graphs via Online Aggregation. In: 38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022. IEEE; 2022. p. 2695-708.

[31]   Atserias A, Grohe M, Marx D. Size Bounds and Query Plans for Relational Joins. In: 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA. IEEE Computer Society; 2008. p. 739-48.

[32]   Veldhuizen TL. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In: Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014; 2014. p. 96-106.

[33]   Letelier A, Pérez J, Pichler R, Skritek S. Static analysis and optimization of semantic web queries. In: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012; 2012. p. 89-100.

[34]   Bigerl A, Conrads L, Behning C, Saleem M, Ngomo AN. Hashing the Hypertrie: Space- and Time-Efficient Indexing for SPARQL in Tensors. In: Sattler U, Hogan A, Keet CM, Presutti V, Almeida JPA, Takeda H, et al., editors. The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings. vol. 13489 of Lecture Notes in Computer Science. Springer; 2022. p. 57-73.

[35]   Guo Y, Pan Z, Heflin J. LUBM: A benchmark for OWL knowledge base systems. J Web Semant. 2005:158-82.

[36]   Conrads F, Lehmann J, Saleem M, Morsey M, Ngomo AN. Iguana: A Generic Framework for Benchmarking the Read-Write Performance of Triple Stores. In: The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II. Springer; 2017. p. 48-65.

[37]   Stardog Union. Stardog; N.D. Retrieved 2023-05-03. `https://docs.stardog.com/`.