Answering SPARQL Queries on the Web of Data through Zero-Knowledge Link Traversal

Pavlos Fafalios
Information Systems Laboratory, FORTH-ICS
Heraklion, Greece
fafalios@ics.forth.gr

Yannis Tzitzikas
Information Systems Laboratory, FORTH-ICS,
and Department of Computer Science,
University of Crete
Heraklion, Greece
tzitzik@ics.forth.gr

ABSTRACT

Link traversal has emerged as a SPARQL query processing method that exploits the Linked Data principles to dynamically discover data relevant for answering a query by dereferencing online Web resources (URIs) at query execution time. While several approaches for such a lookup-based query evaluation method have been proposed, there exists no analysis of the types (patterns) of queries that can be directly answered on the Web of Data through a "zero-knowledge" approach, i.e., without accessing local or remote endpoints and without a-priori knowledge of available data sources. In this paper, we first provide a method for examining if a SPARQL query can be answered through zero-knowledge link traversal and analyse a large corpus of real SPARQL query logs for finding the frequency and distribution of answerable and non-answerable query patterns. Subsequently, we provide an algorithm for transforming answerable queries to SPARQL-LD queries that bypass the endpoints, as well as a method to estimate their evaluation cost which can be useful for deciding on the query execution strategy to follow. We report experimental results about the efficiency of the transformed queries and discuss the benefits and the limitations of this query evaluation method.

CCS Concepts

•**Information systems** → *Query languages;*

Keywords

SPARQL; Link traversal; SPARQL-LD; Web of Data; Linked Data; Query execution cost

1. INTRODUCTION

The Linked Data principles has enabled the extension of the Web with a global data space based on open standards and protocols, the so-called Web of Data [19]. The current most common way to query this constantly increasing body of knowledge is through SPARQL, where clients send queries to local or remote servers through SPARQL endpoints [9].

However, the low reliability of SPARQL endpoints is the major bottleneck that deters the exploitation of these knowledge bases by real applications [35, 5]. Publicly available endpoints are not optimised for efficiency and they often do not serve many concurrent requests in order to avoid server overloading. Buil-Aranda et al. [5] tested 427 public endpoints and found that their performance can vary by up to 3-4 orders of magnitude, while only 32.2% of public endpoints can be expected to have (monthly) uptimes of 99-100%. In general, SPARQL servers are expensive to host and maintain, while providing a reliable public endpoint is challenging. On the contrary, the Linked Data principles provide a simple publishing method which is based on robust web protocols (HTTP, IRI) and can be easily included in existing publishing workflows, e.g., through content negotiation or RDFa.[1] Thus, there arises the need of alternative, less demanding methods to query Web data [17, 35].

Link traversal, in particular, is a query processing method which relies on the Linked Data principles to answer a query by dereferencing (resolving) online web resources (URIs) dynamically, i.e., at query execution time [15, 17]. Inspired by this line of research, in this paper we focus on the query types that can be directly answered through link traversal, without accessing local or remote endpoints and without considering a starting graph or seed URIs for starting the link traversal. Such a *zero-knowledge* query evaluation method is in line with the dynamic nature of the Web, motivates decentralisation, and enables answering queries without requiring data providers to setup and maintain costly endpoints. Figure 1 positions this query execution method in the axis of the existing interfaces that allow querying Web data. Zero-knowledge link traversal offers high data availability and bandwidth, and low cost of server setup and maintenance, however it limits the supported query capabilities. On the contrary, relying on servers offers almost unrestricted query answering, however the server cost is high and the availability and bandwidth low.

In this paper, we investigate the types of queries that can be answered through zero-knowledge link traversal. We first provide a method for checking if a SPARQL query that is to evaluated on a SPARQL endpoint can be answered without accessing the endpoint. We call such a query *Linked Data-answerable Query (LDaQ)*. Then, we analyze a large corpus

[1]`https://www.w3.org/TR/html-rdfa/`

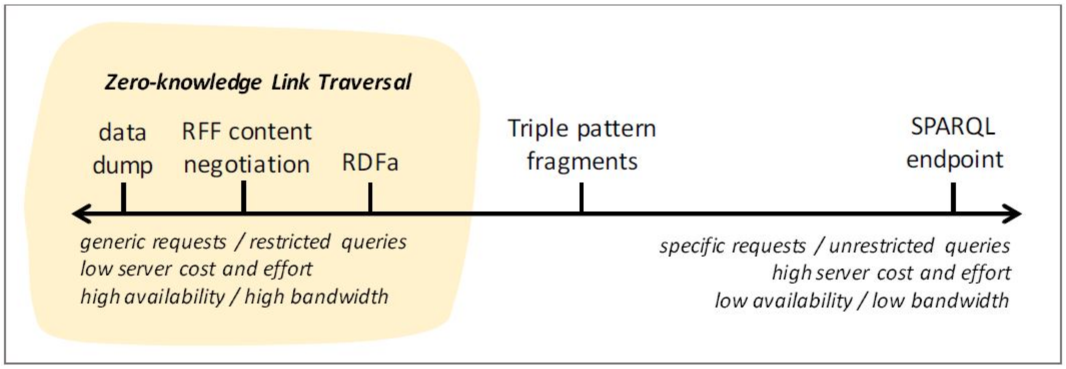
Zero-knowledge Link Traversal
data dump
RFF content negotiation
RDFa
Triple pattern fragments
SPARQL endpoint
generic requests / restricted queries
low server cost and effort
high availability / high bandwidth
specific requests / unrestricted queries
high server cost and effort
low availability / low bandwidth

Figure 1: Interfaces that allow querying Web data and positioning of zero-knowledge link traversal (the figure is a variation of the figure in [33]).

of real query logs from known SPARQL endpoints and study the patterns and frequency of both LDaQs and non-LDaQs. We find that more than 85% of the examined queries are potentially LDaQ, while the majority of them (>84%) follow a few patterns (≤10). Subsequently, we provide an algorithm for transforming LDaQs to SPARQL-LD queries that bypass the endpoints. SPARQL-LD [6] is a SPARQL 1.1 extension (generalization) that enables querying any HTTP resource containing RDF data. We experimentally evaluate the efficiency of the transformed queries and discuss the limitations of this query execution method. We find that more than half of the examined queries can be answered in less than one second, however for queries with large number of intermediate bindings the query execution time can become prohibitively high, thus calling for optimisation methods. Finally, we provide an algorithm to approximate the cost of running a LDaQ, which can be very useful for deciding on the overall query execution strategy to follow. We consider as cost the number of remote resources that need to be accessed since this affects both the query execution time and the amount of data that need to be transferred over the network.

In a nutshell, this paper makes the following contributions:[2]

- A method to examine if a SPARQL query is answerable through zero-knowledge link traversal.
- A method to transform an answerable query to a SPARQL-LD query that is executed through zero-knowledge link traversal.
- A method to estimate the query-execution cost of an answerable query.
- Interesting findings about the type and distribution of frequent answerable and non-answerable queries.
- Experimental results about the efficiency of answerable queries.

The implementation of all algorithms and methods described in this paper, as well as the derived data (LDaQ and non-LDaQ patterns), are publicly available as open source.[3]

The rest of this paper is organised as follows: Section 2 presents the related literature. Section 3 motivates our work and describes the problem and the requirements. Section 4 introduces the methods for finding LDaQ and transforming them to SPARQL-LD queries. Section 5 studies the cost of running LDaQs and provides an algorithm to estimate it. Section 6 presents experimental results. Finally, Section 7 concludes the paper and discusses interesting directions for future research.

2. RELATED WORK

There are three main paradigms for querying distributed RDF data provided by different Web sources: i) data centralisation, ii) query federation, and iii) link traversal.

2.1 Data centralisation

The idea of *data centralisation* is to provide a query service over a collection of cross-domain or domain-specific RDF data gathered (and maybe transformed) from different sources [26, 30]. The current de-facto way for querying such repositories is through SPARQL, where a client submits a SPARQL query to an RDF server through a SPARQL endpoint and the server executes the query and returns the results back to the client. Although data centralisation can provide fast responses, it does not exploit the dynamic nature of Web data (the query results may not reflect the more recent data), and it comes at the cost of setting up and maintaining a centralised repository.

A different approach has been proposed in [35, 33] where the authors introduced the idea of *Triple Pattern Fragments*, a publishing framework that allows efficient offloading of SPARQL query execution from servers to clients. This framework enables servers to maintain high availability rates, allowing querying to scale reliably to much larger numbers of clients. On the downside, the framework requires the setup and maintenance of dedicated servers and clients.

Contrary to this line of research, in this paper we focus on *zero-knowledge* query execution methods that consider the

[2]The paper is an extended version of [7]. It contains an entire new section on estimating the query execution cost of LDaQs (Sect. 5) as well as more details in all other sections.

[3]`https://github.com/fafalios/LDaQ`

full potential of the Web and treat a query in isolation, i.e., the input is only a SPARQL query and no other information on how to answer the query is provided (like the URI of an endpoint or resource).

2.2 Query federation

The idea of *query federation* is to provide integrated access to distributed RDF sources on the Web. DARQ [25] and SemWIQ [21] are two of the first systems to support query federation for SPARQL. Such systems use a mediator service that transparently distributes the execution of queries to multiple endpoints. [28] provides a comprehensive analysis and comparison of a large number of endpoint federation systems. Given the need to address query federation, the SPARQL W3C working group proposed a query federation extension for SPARQL 1.1 [4]. The extension defines the `SERVICE` operator which can be used for executing a graph pattern to a remote endpoint. [4] describes the syntax of this extension and formalizes its semantics. Similar to data centralisation, query federation requires the data to be available through SPARQL endpoints.

SPARQL-LD [6, 8] is a generalisation of SPARQL 1.1 which extends the applicability of the `SERVICE` operator to enable querying any HTTP web source containing RDF data, like online RDF files or web pages embedded with RDFa, JSON-LD, or Microformats.[4] An important characteristic of SPARQL-LD is that it does not require the named graphs to have been declared, thus one can query datasets returned by a portion of the query, i.e., whose URI derives at query execution time. [37] proposes a set of heuristics-based query reordering methods for optimizing the execution of federated queries in both SPARQL 1.1 and SPARQL-LD.

In this paper we make use of SPARQL-LD for transforming a query (to be evaluated on an endpoint) to a SPARQL-LD query that bypasses the endpoint.

2.3 Link traversal

Link traversal exploits the Linked Data principles [19] to dynamically discover data relevant for answering a query [15]. The work in [16, 14] follows RDF links between data sources based on URIs in the query and in partial results. The URIs are resolved over the HTTP protocol into RDF data which is continuously added to the queried dataset using an iterator-based pipeline. Diamond [23] is a similar in spirit query engine to evaluate SPARQL queries through link traversal. [13] studies how the evaluation order in link traversal affects the size of the results and the query execution cost, and proposes a heuristics-based method to optimize query execution. [3], [14] and [12] discuss the notion of *completeness* and propose semantics to restrict the range of link traversal queries.

Index-based link traversal approaches rely on pre-built indexes for finding sources to look up during query execution [11, 29, 36]. [11, 32] uses a combined description of instance- and schema-level elements to summarize the content of data sources. [29] proposes to summarize and index sources based on frequently appearing query graph patterns mined from query logs, while [36] proposes a top-k approach for the index-based source selection strategy, where partial results are located at different sources and can only be accessed via URI lookups. The advantage of index-based approaches is their ability to determine all potentially relevant URIs at the beginning of query execution, which enables to fully parallelize the data retrieval process. However, it comes at the cost of initializing and maintaining the indexes.

[4]`https://github.com/fafalios/sparql-ld`

Regarding more recent works, [31] studies the effectiveness of link traversal-based query execution and proposes reasoning extensions to help finding additional answers. [10] introduces a declarative navigational language for Linked Data, called NautiLOD, which enables to specify data sources by combining navigation and querying capabilities, and also features a mechanism to specify actions that obtain their parameters from data sources reached during the navigation. Finally, LDQL [18] is a more expressive declarative language to query Linked Data through link traversal. LDQL separates the selection of query-relevant regions of Linked Data from the specification of the query result that has to be constructed from the data in the selected regions.

2.4 Positioning

In our work, we focus on *link traversal* and study the types of queries that can be answered directly on the live Web of Data. The starting point of link traversal is *only* the URI(s) that exist in the query's graph pattern and additional URIs are dereferenced only if this is needed for satisfying a triple pattern, i.e. for binding its variables. This means that, in our case, if the query does not contain URIs, it cannot be evaluated through link traversal. This query evaluation method, called *zero-knowledge link traversal*, corresponds to the *query-reachable* completeness class as introduced in [12].

To our knowledge, our work is the first that i) provides open source methods to detect answerable queries and transform them to SPARQL-LD queries that are evaluated without accessing endpoints or indexes, and ii) analyses real query logs from known endpoints for finding popular patterns of queries that can be answered or cannot be answered through zero-knowledge link traversal. While recent works have conducted extensive analytical studies on the syntactical and structural characteristics of real SPARQL queries [1, 27, 2], no previous work has analysed queries in terms of their answerability through link traversal.

3. MOTIVATION, PROBLEM AND REQUIREMENTS

Our objective is to study the type of SPARQL queries that can be directly executed on the live Web of Data, without a priori knowledge of available data sources. The motivation for this zero-knowledge (or query-reachable) approach is threefold:

- *Freshness.* The dynamic nature of the Web of Data which constitutes a huge and constantly evolving information space, meaning that we may always need to query a new (unknown) resource not existing in our repository, index or seed URIs.
- *Convenience.* The capability to easily run queries directly on the Web of Data, from any client that sup-

ports SPARQL, without the need to build and maintain indexes and without requiring data providers to setup and maintain costly servers.

- *Decentralisation.* the Web of Data is increasingly becoming a centralised data space relying on server-side infrastructures [34]. Enabling the execution of SPARQL queries directly on the Web of Data can motivate more publishers to put their data online (e.g., by uploading RDF files), since their data becomes queryable and exploitable without putting effort on setting up and maintaining reliable servers.

Consider, for example, the query in Figure 2 which requests the birth date of Barack Obama, and the SPARQL endpoint of DBpedia which can provide an answer to this query. Our aim is to answer the query without accessing DBpedia's endpoint. One approach is to access the URI of Barack Obama used in the query, retrieve the triples contained in this URI, and then run the corresponding triple pattern on these triples. Figure 3 shows a SPARQL-LD query that achieves this. The query uses the extended `SERVICE` operator of SPARQL-LD to retrieve and query the triples that are contained in the URI of Barack Obama, thereby bypassing DBpedia's endpoint. However, to apply such a transformation, the URI must be dereferenceable and return all the outgoing properties of the corresponding entity.

```
SELECT ?birthDate WHERE {
  dbr:Barack_Obama dbo:birthDate ?birthDate }
```

Figure 2: Example of a LDaQ requesting the birth date of Barack Obama.

```
SELECT * WHERE {
  SERVICE dbr:Barack_Obama {
    dbr:Barack_Obama dbo:birthDate ?birthDate } }
```

Figure 3: The transformed SPARQL-LD query of the query in Figure 2.

The query in Figure 4 requests the birth date of all basketball players in DBpedia. In this case, to be able to bypass DBpedia's endpoint, the URI of the DBpedia class *Basketball Player* must contain all its incoming properties, i.e. the instances of the class *Basketball Player*. The query in Figure 5 shows the corresponding SPARQL-LD query. The query first accesses the URI of the DBpedia class *Basketball Player* to retrieve its instances, and then accesses the URI of each instance to retrieve the values of the birth date property.

However, not all queries can be transformed to SPARQL-LD queries. Figure 6 shows two such queries. The first query requests all things (of unknown type) having the name *"Michael Jordan"*, while the second requests the total number of triples. The first query could bypass the endpoint if the URI of the `foaf:name` property provided all the triples that contain it as predicate. However, this is not common in Linked Data and also impractical for large datasets and popular properties (like `rdf:type` and `rdfs:label`). With respect to the second query, we consider the entire Web of Data as a single RDF repository of unknown number of triples. Thereby, in our work we are not interested in queries that require access to the full repository.

```
SELECT ?player ?birthDate WHERE {
  ?player rdf:type dbo:BasketballPlayer ;
          dbo:birthDate ?birthDate }
```

Figure 4: Example of a LDaQ requesting the birth date of all basketball players in DBpedia.

```
SELECT ?player ?birthDate WHERE {
  SERVICE dbo:BasketballPlayer {
    ?player rdf:type dbo:BasketballPlayer }
  SERVICE ?player {
    ?player dbo:birthDate ?birthDate } }
```

Figure 5: The transformed SPARQL-LD query of the query in Figure 4.

```
SELECT ?entity WHERE {
  ?entity foaf:name "Michael Jordan" }
```

```
SELECT COUNT(*) WHERE {
  ?s ?p ?o }
```

Figure 6: Example of non-LDaQs.

We now define two requirements that can enable this functionality for a large portion of SPARQL queries:

- **R1:** URIs must be dereferenceable and return RDF data.
- **R2:** URIs must provide both the incoming and outgoing properties of the corresponding resource, i.e., all triples in which the URI is the *subject* or the *object* in the triple. This includes URIs that represent RDFS/OWL classes, meaning that the URI of a class should return all its instances.

These requirements are in line with the Linked Data principles [19]. An obvious problem of R2 is when the URI represents classes, since the number of instances can be very large for generic classes (like *Person* or *Location*). We discuss this case at Section 6.3.

4. FINDING & TRANSFORMING LINKED DATA-ANSWERABLE QUERIES

In this section, we define the notion of Linked Data-answerable query (Section 4.1), provide algorithms for checking if a graph pattern or query is Linked Data-answerable (Section 4.2), introduce a method to transform answerable queries to SPARQL-LD queries that bypass the endpoints (Section 4.3), and finally discuss special cases (Section 4.4) as well as problems and limitations (Section 4.5).

The implementation of all algorithms described in this section is publicly available as open source (see Footnote 3).

4.1 Linked Data-answerable graph patterns and queries

Following the definitions of [24], let first $\mathcal{U}$ be an infinite set of URIs, $\mathcal{B}$ an infinite set of blank nodes and $\mathcal{L}$ an infinite set of literals. The union of these sets constitutes the set of *RDF terms*. A triple $(s, p, o) \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is called an *RDF triple*, where s is the *subject*, p is the *predicate* and o is the *object*. We denote by $s(t)$, $p(t)$ and $o(t)$, the subject, predicate and object, respectively, of a triple t. Let also $\mathcal{V}$ be a set of variables that can bind to RDF terms from $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. A triple $p \in (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})$ is called *triple pattern*, while a *Basic Graph Pattern (BGP)* is a set of triple patterns. Finally, let $\mathcal{V}_i^b$ be the set of bound variables *before* the execution of the i-th triple pattern of a BGP.

We now define the notion of *Linked Data-answerable BGP*:

Definition 1. A BGP is Linked Data-answerable, for short LDaBGP, if its triple patterns T can be brought into an order such that each triple contains at least one URI or bound variable, i.e.: $\forall t_i \in T,\ s(t_i) \in \mathcal{U} \lor o(t_i) \in \mathcal{U} \lor s(t_i) \in \mathcal{V}_i^b \ \lor\ o(t_i) \in \mathcal{V}_i^b$.

This definition corresponds to the *query-reachable* completeness class and the *completely-answerable* BGPs as introduced in [12].

Queries containing one or more UNION groups need special handling. Through this operator, SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. Consider, for example, the query in Figure 7 which requests the birth date and place of basketball and football players. The query contains two UNION groups, each one containing two UNION graph patterns. Moreover, the query contains two triples that are not part of the UNION groups (line 4). To decide if such a query is Linked Data-answerable, we must check all the graph patterns of each UNION group as well as the triples outside the UNION patterns. However, we should not check them in isolation. For example, the graph patterns of the last UNION group are not Linked Data answerable by themselves, but they are answerable if we consider the bindings of the preceding triples and UNION groups. We first define the notion of Linked Data-answerable UNION group:

Definition 2. A UNION group of BGPs is Linked Data-answerable if each of its BGPs is Linked Data-answerable.

```
1 SELECT ?player ?birthDate ?birthPlaceName WHERE {
2   { ?player rdf:type dbo:BasketballPlayer }
3   UNION { ?player rdf:type dbo:FootballPlayer }
4   ?player dbo:birthDate ?birthDate ; dbo:birthPlace ?place .
5   { ?place foaf:name ?birthPlaceName }
6   UNION { ?place rdfs:label ?birthPlaceName } }
```

Figure 7: Example of a SPARQL query containing two UNION groups.

Now we define the notion of *Linked Data-answerable Query* which contains as elements BGPs and UNION groups.

Definition 3. A SPARQL query containing as elements BGPs (sets of triple patterns) and UNION groups is Linked Data-answerable, for short LDaQ, if its elements can be brought into an order such that each of them is answerable given the variable bindings before the execution of the corresponding BGP / UNION group.

Algorithm 1 isLDaBGP

```
Input: P: graph pattern, inUnion: boolean (optional), B: bound variables (optional)
Output: true or false
 1: LB = {}                                          ▷ Locally-bound variables
 2: M = {}          ▷ Map a var to other vars that can help binding it
 3: if B != NULL then
 4:     LB.addAll(B)
 5: for t : triples(P) do
 6:     if isURI(t.subject) then
 7:         if isVariable(t.object) then LB.add(t.object)    ▷ Object variable can be bound
 8:         if isVariable(t.predicate) then LB.add(t.predicate)    ▷ Predicate variable can be bound
 9:     else if isURI(t.object) then
10:         if isVariable(t.subject) then LB.add(t.subject) ▷ Subject variable can be bound
11:         if isVariable(t.predicate) then LB.add(t.predicate)    ▷ Predicate variable can be bound
12:     else if isVariable(t.subject) & isVariable(t.object) then
13:         if (LB.contains(t.subject) then LB.add(T.object)
14:             if (isVariable(t.predicate) then LB.add(T.predicate)
15:         else if (LB.contains(t.object) then LB.add(T.subject)
16:             if (isVariable(t.predicate) then LB.add(T.predicate)
17:         else
18:             M.add(t.subject, t.object)       ▷ Binding of object variable can bind the subject variable
19:             M.add(t.object, t.subject)       ▷ Binding of subject variable can bind the object variable
20:             if isVariable(t.predicate) then
21:                 M.add(t.predicate, {t.subject, t.object})
22: V = getAllVariables(P)        ▷ Set of all graph pattern variables
23: for v : V do       ▷ Check for any variable that cannot be bound
24:     if v ∉ LB then
25:         if !isBindable(M.get(v)) then
26:             return false       ▷ Recursively check if the var can be bound through the bindings of other vars
27: if !inUnion then
28:     B.addAll(LB);           ▷ Add to B the locally bound variables
29: return true
```

4.2 Checking the answerability of graph patterns and queries

Algorithm 1 provides a method to find out if a basic graph pattern is Linked Data-answerable or not. In brief, the algorithm goes through the triple patterns and finds "bindable" variables, i.e., variables that can get bound by dereferencing a URI that exists in the triple, or that can get bound through bindings of other variables. If at the end there is at least one non-bindable variable, then the query is not a LDaQ. The algorithm can also be provided with two optional parameters. The parameter B (bound variables) enables to provide a set of already-bound variables, which is useful for cases where the input graph pattern is part of a query. The parameter *inUnion* allows specifying that the input graph pattern is part of a UNION group, thus its bindings must not be considered when checking the other UNION graph patterns of the same UNION group.

Algorithm 2 checks if a query is Linked Data-answerable or not. Each triple and UNION group in the query is considered a different *element*. The algorithm first goes through all the query's elements and checks their answerability using Algorithm 1. In case the element is a UNION group the algorithm checks the answerability of each UNION's graph pattern. If the element is not answerable, it is added to a list of pending elements (since they may be answerable when some variables in another element get bound). If the element is answerable, the list of bound variables is updated

Algorithm 2 isLDaQ

```
Input: Q: Query graph pattern
Output: true or false
 1: B = {}                                      ▷ Bound variables
 2: PEND = {}                           ▷ Pending query elements
 3: E = getElements(Q)    ▷ Each triple and each UNION group is
    considered a different element
 4: for e : E do                  ▷ For each triple or UNION group
 5:     if isTripleElement(e) then          ▷ The element is a triple
 6:         if !isLDaBGP(e, false, B) then
 7:             PEND.add(e)   ▷ Add this element to the pending list
 8:     else                        ▷ The element is a UNION group
 9:         allAnswerable = true
10:         for u : getUnionGraphPatterns(e) do
11:             if !isLDaBGP(u, true, B) then
12:                 allAnswerable = false;
13:                 break;
14:         if allAnswerable then B.addAll(getVariables(e))
15:         else PEND.add(e)
16: while !PEND.isEmpty() do
17:     foundNew = false    ▷ In each FOR loop we must find a new
    answerable element
18:     for pe : PEND do                  ▷ For each pending element
19:         if isTripleElement(pe) then
20:             if isLDaBGP(pe, false, B) then
21:                 foundNew = true; PEND.remove(pe)
22:         else            ▷ The pending element is a UNION group
23:             allAnswerable = true
24:             for u : getUnionGraphPatterns(pe) do
25:                 if !isLDaBGP(u, true, B) then
26:                     allAnswerable = false;
27:                     break;
28:             if allAnswerable then
29:                 foundNew = true
30:                 PEND.remove(pe)
31:                 B.addAll(getVariables(pe))
32:     if !foundNew then
33:         return false      ▷ No new answerable element was found
34: return true                 ▷ The query is Linked Data-answerable
```

with the element's variables. Then, the algorithm checks the pending elements. In each loop, at least one new element must get answerable, otherwise the query is not Linked Data-answerable.

4.3 Transforming answerable queries to SPARQL-LD

We now provide a method to transform a LDaQ to a SPARQL-LD query that evaluates its graph pattern directly over the live Web of Linked Data without accessing local or remote endpoints. Such a *transformation-based* approach to run LDaQs offers the ability to directly make use of this query execution method through existing instances of SPARQL-LD, i.e., without the need to setup a dedicated server that supports the execution of link traversal queries.

Algorithm 3 transforms a BGP to a SPARQL-LD graph pattern. The algorithm goes through the triples and creates SERVICE patterns. Specifically, if the triple contains a URI or a bound variable, it checks if there is already a SERVICE pattern for the same URI/ variable. If so, the triple is just added to its graph pattern, otherwise a new SERVICE pattern is created. Notice that if both the subject and object are URIs, the algorithm looks up only the subject URI (but one could resolve both). If the triple does not contain a URI or bound variable, it is added to a list of pending triples. Since the BGP is Linked Data-answerable, these triples require the binding of another variable existing in a subsequent triple. After checking all triple patterns, the algorithm goes through the pending triples and, correspondingly, creates new SERVICE patterns or updates the existing ones.

Algorithm 4 transforms a Linked Data-answerable query (that may also contain UNION groups) to a SPARQL-LD query. The algorithm goes through the query's elements (which can be either single triples or UNION groups) and checks if they are Linked Data-answerable. If so, the procedure 'include' is executed (Procedure 1). This procedure includes the element to the SPARQL-LD query, either by appending it to an existing SERVICE or by creating a new one. If the element is not Linked Data-answerable, it is added to a list of pending elements whose transformation requires the binding of a variable existing in a subsequent triple or UNION group. Then the algorithm goes through the pending elements and includes them in the transformed SPARQL-LD query once they get answerable.

Algorithm 3 transformBGP

```
Input: Basic graph pattern P
Output: SPARQL-LD query pattern P'
 1: P' = {}                            ▷ SPARQL-LD graph pattern
 2: B = {}                                     ▷ Bound variables
 3: PEND = {}                           ▷ Pending triple patterns
 4: for t ∈ getTriples(P) do               ▷ For each triple pattern
 5:     if isURI(t.subject) || isURI(t.object) then
 6:         u = isURI(t.subject) ? t.subject : t.object    ▷ Consider
    subject or object URI
 7:         if P'.containsService(u) then ▷ There is a service pattern
    for the same URI
 8:             P'.getService(u).add(t)  ▷ Add the triple pattern to its
    graph pattern
 9:         else P'.add(new ServicePattern(u, t))      ▷ Create a new
    service pattern
10:         updateBoundVariables(t, B)    ▷ Update the set of bound
    variables
11:     else if B.contains(t.subject) || B.contains(t.object) then
12:         v = B.contains(t.subject) ? t.subject : t.object
13:         if P'.containsService(v) then
14:             P'.getService(v).add(t)
15:         else P'.add(new ServicePattern(v, t))
16:         updateBoundVariables(t, B)
17:     else PEND.add(t)          ▷ Transform this triple pattern later
18: while !PEND.isEmpty() do
19:     for pt ∈ PEND do               ▷ For each pending triple pattern
20:         if B.contains(pt.subject) || B.contains(pt.object) then
21:             v = B.contains(pt.subject) ? t.subject : t.object
22:             if P'.containsService(v) then
23:                 P'.getService(v).add(pt)
24:             else P'.add(new ServicePattern(v, pt))
25:             updateBoundVariables(pt, B); PEND.remove(pt)
26: return P'
```

4.4 Special cases

In this paper we do not examine the case of DESCRIBE queries, as well as of queries containing the operators FROM, FROM NAMED / GRAPH, and SERVICE. These queries correspond to around 15% of the queries submitted to popular SPARQL endpoints [1]. Indicatively, for DESCRIBE queries we can just look up the provided URI and return all its triples. For FROM and FROM NAMED / GRAPH queries, the triples of the provided resource should be fetched and the corresponding graph pattern can be directly executed over these triples (without requiring to check its answerability). Finally, SERVICE patterns over remote endpoints can be also transformed to SPARQL-LD queries if their graph pattern is Linked Data-answerable. We leave

Algorithm 4 transformQuery

```
Input: Q: query graph pattern
Output: Q': SPARQL-LD query pattern
 1: Q' = {}; B = {}; PEND = {}    ▷ SPARQL-LD pattern; Bound vars; Pending elements
 2: E = getElements(Q)    ▷ Each triple and each UNION group is considered a different element
 3: for e : E do    ▷ For each triple or UNION group
 4:     if isTripleElement(e) then    ▷ The element is a triple
 5:         if isLDaBGP(e, false, B) then
 6:             include(Q', e, B)
 7:         else PEND.add(e)    ▷ Transform it later
 8:     else    ▷ The element is a UNION group
 9:         allAnswerable = true
10:         for u : getUnionGraphPatterns(e) do
11:             if !isLDaBGP(u, true, B) then
12:                 allAnswerable = false;
13:                 break;
14:         if allAnswerable then
15:             B.addAll(getVariables(e))
16:             include(Q', e, B)
17:         else PEND.add(e)    ▷ Transform it later
18: while !PEND.isEmpty() do    ▷ While there exist pending elements
19:     for pe : PEND do    ▷ For each pending element
20:         if isTripleElement(pe) then
21:             if isLDaBGP(pe, false, B) then
22:                 include(Q', e, B);
23:                 PEND.remove(pe)
24:         else    ▷ The pending element is a UNION group
25:             allAnswerable = true
26:             for u : getUnionGraphPatterns(pe) do
27:                 if !isLDaBGP(u, true, B) then
28:                     allAnswerable = false; break;
29:             if allAnswerable then
30:                 include(Q', e, B)
31:                 PEND.remove(pe); B.addAll(getVariables(pe))
32: return Q'
```

Procedure 1 The 'include' procedure

```
 1: procedure INCLUDE(Q, e, B)
 2:     if isTripleElement(e) then    ▷ The element is a triple
 3:         if isURI(e.subject) || isURI(e.object) then
 4:             u = isURI(e.subject) ? e.subject : e.object    ▷ Consider the subject or object URI
 5:             if Q.containsService(u) then
 6:                 Q.getService(u).add(e)    ▷ Add the triple to the graph pattern of the existing SERVICE
 7:             else
 8:                 Q.add(new ServicePattern(u, e))    ▷ Create a new SERVICE pattern
 9:         else if e.subject in B || e.object in B then
10:             v = e.subject in B ? e.subject : e.object    ▷ Consider the subject or object variable
11:             if Q.containsService(v) then
12:                 Q.getService(v).add(t)    ▷ Add the triple to the existing SERVICE graph pattern
13:             elseQ.add(new ServicePattern(v, t))    ▷ Create a new SERVICE pattern
14:     else    ▷ The element is a UNION group
15:         v = commonBoundVariable(e, B, Q)    ▷ Find the common bound variable
16:         if v = NULL then    ▷ Case of URI: create a UNION of SERVICE patterns
17:             U' = new UnionGroup()
18:             for u : getUnionGraphPatterns(e) do
19:                 U'.addUnionPattern(transformBGP(u))
20:             Q.add(U')
21:         else    ▷ Case of variable: append it to an existing SERVICE or create a new one
22:             if Q.containsService(v) then
23:                 Q.getService(v).add(e)
24:             else
25:                 Q.add(new ServicePattern(v, e))
```

the implementation of all these cases as part of our future work.

4.5 Problems and limitations

There are some data access issues that must be taken into account when running queries over the live Web of Linked Data. The work in [15] (Section 2) discusses the main problems and limitations. In brief, dereferencing a URI may result in the retrieval of an unforeseeable large set of RDF triples, while response times may vary significantly between different web servers and time periods. Moreover, some servers put restrictions on clients such as serving only a limited number of requests per second. At any case, a Linked Data query execution system should implement a politeness policy to avoid overloading servers, by respecting, for example, the *robots.txt* protocol[5] that allows web sites to demand delays between subsequent requests from the same client, or by using a default minimum delay between consecutive requests on the same server [20].

5. COST ESTIMATION

We now study the cost of running LDaQs. We consider *query cost* as the number of remote resources that have to be accessed during query execution. This number affects both the query evaluation time as well as the amount of data that is transferred through the network. Below, we first discuss the most common query characteristics that affect the query cost (Section 5.1) and then provide an algorithm to estimate it (Section 5.2).

5.1 Characteristics affecting the query cost

Given a basic graph pattern, the first characteristic that affects the query cost is the number of distinct URIs that appear as subjects or objects in the graph pattern's triples. These URIs have to be resolved during query evaluation for either binding variables or making the necessary joins.

The second characteristic is the number of distinct variables whose bindings can bind other variables. For each such variable, the query needs to resolve all its URI bindings for evaluating the graph pattern, which can be very costly for cases of large number of bindings. We call these variables *necessary-to-resolve* variables. Consider, for example, the query in Fig. 8. The query requests all authors together with the venues of their publications. The query first has to access the URI of the `:Author` class for binding the variable `?author`. The number of bindings can be high. Then, the query needs to access the URI of each author for binding the variable `?publication`. Finally, the query needs to access the URI of each author publication for binding the variable `?venue`. If we consider that the number of authors returned by the `:Author` URI is around 10,000 and that each author has, on average, 50 publications, then the total number of URIs that need to be accessed for evaluating the query is 510,001 (1 for binding the variable `?author` + 10,000 for binding the variable `?publication` of each author + 10,000×50 for binding the variable `?venue` of each author publication).

[5] `http://www.robotstxt.org/`

```
1 SELECT ?author ?venue WHERE {
2   ?author a :Author .
3   ?author :hasPublication ?publication .
4   ?publication :inVenue ?venue  }
```

Figure 8: Example of SPARQL query.

The third characteristic that affects the query cost is the type and value of the predicate used to bind a necessary-to-resolve variable. First, if the predicate is a variable, then the number of bindings of the necessary-to-resolve variable can be very high since all different predicates connecting the subject with the object are considered. The query of Fig. 9, for instance, requests all URI properties of authors (i.e., their related entities) together with their labels. The number of bindings of the variable `?relatedEntity` (which needs to be resolved for binding the variable `?label`) can be very high since all properties of all authors are considered. If now the predicate is a URI, its value can affect the number of bindings of the corresponding necessary-to-resolve variable. There are predicates for which the objects have, on average, small number of subjects but also predicates for which the objects can have a very large number of subjects. For instance, we know that the capital of a country can be only one, thus for the `:hasCapital` predicate we expect one object binding. On the contrary, a very experienced researcher usually has a large number of publications, thus for the `:hasPublication` predicate we can expect a large number of object bindings. Likewise, we can expect a small number of subject bindings for some predicates and a large number for some other. For instance, on average we expect a large number of subject bindings for the predicate `birth place` (there are many persons with the same birth place), but a small number for the predicate `has capital` (there is only one country for a given capital city).

```
1 SELECT * WHERE {
2   ?author a :Author .
3   ?author ?property ?relatedEntity .
4   ?relatedEntity :label ?label }
```

Figure 9: Example of SPARQL query.

The fourth characteristic is the number of star-shaped joins that limit the bindings of necessary-to-resolve variables (which, in this case, are the common variables in the joins). For example, consider the query in Fig. 10. Compared to the query of Fig. 8, this query contains one additional triple pattern (line 3) which limits the bindings of the variable `?author` to only those having a `:directorOf` property. This triple pattern forms a star-shaped join with the first triple pattern (line 2), requesting only authors who have a director position at an institution. In this case, the number of bindings of the variable `?author` can be highly reduced. For example, if the number of authors in the knowledge base who are directors is 100, then the total number of URIs that need to be accessed for evaluating the query is limited from 510,001 to 15,101 (1 for binding the variable `?author` + 10,000 for checking the `directorOf` property of all authors + 100 for binding the variable `?publication` of each author who is also director + 100×50 for binding the variable `?venue` of each author publication).

```
1 SELECT ?author ?venue WHERE {
2   ?author a :Author .
3   ?author :directorOf ?institution .
4   ?author :hasPublication ?publication .
5   ?publication :inVenue ?venue  }
```

Figure 10: Example of SPARQL query.

The number of bindings can be also reduced if the necessary-to-resolve variable is involved in a chain-shaped join whose other subject or object elements are URIs or literals. Consider, for example, the query in Fig. 11. The query is the same with that of Fig. 8 with an addition of one triple (line 1) which requires the authors to belong to a specific party. The two triples in lines 1 and 2 form a chain-shaped join which limits the number of bindings of the variable `?author`.

```
1 SELECT ?author ?venue WHERE {
2   :party12 :hasMember ?author .
3   ?author a :Author .
4   ?author :hasPublication ?publication .
5   ?publication :inVenue ?venue  }
```

Figure 11: Example of SPARQL query.

Another query characteristic that can highly limit the bindings of a necessary-to-resolve variable and thus the query cost is the appearance of FILTER operators. The FILTER operator is a constraint which restricts the solutions (variable bindings) of the whole group of triple patterns in which it appears. Consider, for example, the query in Fig. 12. The query is the same with that of Fig. 10 with an addition of one triple pattern and one FILTER operator (line 3). The triple pattern requests the birth date of each author while the filter operator restricts the accepted values of the `?birthDate` bindings to only those of year after 1985. Since the `?author` variable exists in the same triple with the `?birthDate` variable, the filter operator can highly limit its bindings. For instance, if the number of authors who are directors and have a birth date after 1985 is 10, then the total number of URIs that need to be accessed is further limited to 10,511 (1 for binding the variable `?author` + 10,000 for checking the `directorOf` property of all authors + 10 for binding the variable `?publication` of each author who is director and has a birth date after 1985 + 10×50 for binding the variable `venue` of each author publication).

```
1 SELECT ?author ?venue WHERE {
2   ?author a :Author .
3   ?author :directorOf ?institution .
4   ?author :birthDate ?birthDate . FILTER(year(?birthDate) > 1985)
5   ?author :hasPublication ?publication .
6   ?publication :inVenue ?venue  }
```

Figure 12: Example of SPARQL query.

A last characteristic that can affect the query cost is the order of the triples and FILTERs in the graph pattern. Query writing is not always optimal and this can affect the execution time of a query if the underlying SPARQL implementation does not apply a query optimisation technique, e.g., a query re-ordering method [37]. For example, in the query of Fig. 12, if we move the second and third triple to the end, the query cost is highly increased (considering that no query reordering is applied), since the query needs to first retrieve the venues of all publications of all 10,000 authors before restricting the bindings of the `?author` variable.

Below we describe an algorithm to estimate the query cost based on the abovementioned characteristics of a query graph pattern.

5.2 Estimating the query execution cost

Algorithm 5 estimates the query execution cost of a basic graph pattern.[6] The algorithm goes through the triples of the graph pattern and first checks if the subject or object of the triple is a URI (line 7). In this case the cost is increased if the URI has not been previously resolved. If the other elements of the triple are variables, then we add them in the list of bound variables (function '`addVar`') together with additional information about the triple required for better cost estimation, in particular: i) the predicate used to bind the variable, ii) the position of the variable in the triple ('subject', 'object', or 'predicate'), and ii) the element of the triple that bound the variable (lines 12-19). If both the subject and the object of the triple are variables (line 20), then the URI bindings of the subject (or object) variable need to be resolved (if not yet) for binding the object (or subject) and predicate variables. In this case, we need to estimate the expected number of bindings for that necessary-to-resolve variable using the function '`estimBindingsNum`' (we detail this function below) and increase the cost accordingly (lines 24 and 31). The same needs to be done if the subject is a variable and the object a literal (lines 39-45), since this is required in order to perform the corresponding joins. If both the subject and the object variables are not bound (line 23), then we skip the triple by adding it to a list of pending triples (since the query is a LDaQ, one of the variables will get bound in a subsequent triple). We do the same if the subject of the triple is an unbound variable and the object a literal (lines 46-47). If both the subject and the object variables are bound (line 37) then no action is required (no need to resolve any URI bindings). Finally, the algorithm goes through the pending triples (lines 48-50) and repeats the same procedure. At each loop, at least one of the pending triples is expected to be removed from this list (since the query is a LDaQ).

Estimating the number of variable bindings. To estimate the expected number of bindings of a necessary-to-resolve variable (function '`estimBindingsNum`' in Alg. 5), we need to consider the previous estimations of other necessary-to-resolve variables. For example, as we say in the query of Fig. 8, for estimating the maximum number of bindings of the variable `?publication` (whose URI bindings should be resolved for binding the variable `?venue`), we need to know the estimated number of bindings of the variable `?author` and multiple it with the expected number of object bindings of the predicate `:hasPublication`. If the number of authors is 100 and each author has 10 publications on average, the estimated number of publications that need to be accessed is 1,000 (in the worst case where no authors share the same publications). To accelerate this process, we can pre-compute the average number of subject

[6]The implementation of the algorithm is publicly available (see Footnote 3).

Algorithm 5 estimateCost

```
Input: P: query graph pattern
Output: cost: query cost
 1: B = {}                                              ▷ Bound variables
 2: U = {}                                              ▷ Resolved URIs
 3: R = {}                       ▷ Variables whose URIs have been resolved
 4: PEND = {}                                           ▷ Pending triples
 5: cost = 0
 6: for t : triples(P) do
 7:     if isURI(t.subject) || isURI(t.object) then ▷ If the subject or object is a URI
 8:         u = isURI(t.subject) ? t.subject : t.object  ▷ Get the URI
 9:         if $u \notin U$ then
10:             c++    ▷ The URI must be resolved, increase the cost
11:             U.add(u)   ▷ Add the URI in the list of resolved URIs
12:         if isVar(t.object) then ▷ If the object is a variable, it can be bound from the subject URI
13:             B.addVar(t.object, t.predicate, "OBJ", t.subject)   ▷ Add the variable to B, together with additional information required for cost estimation
14:             if isVar(t.predicate) then
15:                 B.addVar(t.predicate,-, "PRED", t.subject)
16:         if isVar(t.subject) then   ▷ If the subject is a variable, it can be bound from the object URI
17:             B.addVar(t.subject, t.predicate, "SUB", t.object)
18:             if isVar(t.predicate) then
19:                 B.addVar(t.predicate,-, "PRED", t.object)
20:     else if isVar(t.subject) && isVar(t.object) then   ▷ Both subject and object are variables
21:         if B.contains(t.subject) && !B.contains(t.object) then
22:             if !R.contains(t.subject) then   ▷ All URIs of the subject var should be resolved for binding the object var
23:                 R.add(t.subject)
24:                 cost += estimBindingsNum(t.subject, B)   ▷ Estimate the num of bindings of the necessary-to-resolve variable, considering the previous estimations of other necessary-to-resolve variables, as well as the predicate(s) used to bind this variable and additional information like appearance of joins and FILTERs
25:             B.addVar(t.object, t.predicate, "OBJ", t.subject)
26:             if isVar(t.predicate) then
27:                 B.addVar(t.predicate,-, "PRED", t.subject)
28:         if B.contains(t.object) && !B.contains(t.subject) then
29:             if !R.contains(t.object) then ▷ All URIs of the object variable should be resolved for binding the subject variable
30:                 R.add(t.object)
31:                 cost += estimBindingsNum(t.subject, B)   ▷ Estimate the num of bindings of the necessary-to-resolve variable, considering the previous estimations of other necessary-to-resolve variables, as well as the predicate(s) used to bind this variable and additional information like appearance of joins and FILTERs
32:             B.addVar(t.subject, t.predicate, "SUB", t.object)
33:             if isVar(t.predicate) then
34:                 B.addVar(t.predicate,-, "PRED", t.object)
35:         if !B.contains(t.subject) && !B.contains(t.object) then
36:             PEND.add(t) ▷ The vars will get bound in next triples
37:         if B.contains(t.subject) && B.contains(t.object) then
38:             // nothing; the two variables are already bound
39:     else        ▷ The subject is a variable and the object a literal
40:         if B.contains(t.subject) then
41:             if !R.contains(t.subject) then
42:                 R.add(t.subject)
43:                 cost += estimBindingsNum(t.subject, B)   ▷ Estimate the num of bindings of the necessary-to-resolve variable, considering the previous estimations of other necessary-to-resolve variables, as well as the predicate(s) used to bind this variable and additional information like appearance of joins and FILTERs
44:             if isVar(t.predicate) then
45:                 B.addVar(t.predicate,-, "PRED", t.subject)
46:         else
47:             PEND.add(t)    ▷ The subject var will get bound later
48: while !PEND.isEmpty() do
49:     for t : PEND do
50:         //Repeat lines 7-42 and at each iteration remove from PEND the triple(s) whose all variables exist in B
51: return cost
```

and object values for all known URI predicates. For example, for the predicate `:birthPlace`, we might know that the average number of subjects (persons) for any given object (country) is around 100 (i.e., 100 persons in the knowledge base, on average, were born in a specific country), while the average number of objects (countries) for any given subject (persons) is only 1 (i.e., 1 country is the birth place of a person). Using these pre-computed mappings, we can estimate the expected number of bindings very fast, e.g., at query evaluation time. If computing them is not possible, we can consider a constant number > 1 for all cases. Such an approach provides an estimation considering only the structure of graph pattern, thus it can sometimes provide wrong estimations. If now the predicate is not a URI but a variable, we saw that this highly increases the estimated number of bindings of the corresponding subject or object variable. In this case, we can precompute the average number of predicates per subject or object (or consider an arbitrary number) and multiply it with the average number of subject or object values of all predicates.

For better cost estimation, the function can also consider if the variable forms a join (star- or chain-shaped) in triples of different predicates, where the other elements (subject/object) of the two triples are URIs. In this case, we can consider the *minimum* number of expected bindings of the two predicates. Consider, for instance, the query in Fig. 11. If the number of subjects who belong to at least one party (line 1) is 100 and the number of authors 10,000 (line 2), the maximum number of bindings for the variable `?author` is 100, and thus the query needs to access the publication URIs (lines 3-4) of 100 persons. We can do the same in cases where the variable is involved in star-shaped joins, as explained in the example of query 10. In addition, the function can check for any FILTER operators in the query graph pattern and reduce the expected number of bindings of the involved variables (as explained in the example of query 12). Since it is almost impossible to catch all different cases of FILTER operators and study how they affect the number of bindings of the involved variables, a general approach is to apply a constant reduction function in all cases, e.g., divide it by n, for example $n = 10$, or consider its log value.

Finally, in case of queries containing UNION groups, we need to sum the costs of all union graph patterns and the estimated number of bindings of the involved variables.

6. EXPERIMENTAL RESULTS

6.1 Datasets

We experimented with real SPARQL query logs provided by the Linked SPARQL Queries Dataset (LSQ) [27] and the USEWOD series of workshops [22]. From LSQ, we used all the queries of Linked Geo Data (LGD), Semantic Web Dog Food (SWDF), British Museum (BM), and DBpedia, while from USEWOD we used the queries of LGD, SWDF, BIO2RDF, and the more recent DBpedia 2014 and 2015 queries. The total number of queries in these datasets is 67,849,121.

We first fixed some common errors found in the queries (like the absence of popular prefixes), and then used Jena 3.2 to parse them and get their graph pattern. In our experiments, we did not consider the queries that are not valid according to Jena 3.2 and the queries that use property paths or contain one of the following operators: DESCRIBE, FROM, GRAPH, SERVICE, MINUS, EXISTS, BIND, VALUES, SUB-SELECT (nested queries).

Table 1 shows the main statistics per dataset. The last column shows the total number of *unique* queries that we consider in our analysis. For finding the unique queries, we compared only the query graph patterns, i.e., without considering the prefixes, the SELECT clause, and any ORDER/GROUP BY operators.

6.2 Pattern-based analysis of answerable and non-answerable queries

We examined the Linked Data-answerability of all unique queries (using Algorithm 2) as well as the pattern (template) they follow. For getting the pattern of a query, we considered only its graph pattern (text under WHERE), removed the FILTER operators, and replaced all variables, URIs, literals, and blank nodes with the strings [V], [U], [L], and [B], respectively.[7] For example, the pattern of the query in Figure 7 is the following:

```
{ [V] [U] [U] } UNION { [V] [U] [U] } [V] [U] [V] ; [U] [V]
{ [V] [U] [V] } UNION { [V] [U] [V] }
```

Table 2 shows the number and percentage of LDaQ and non-LDaQ, and the corresponding number of unique patterns. We notice that the percentage of LDaQ is more than 85% in all datasets. BM and BIO2RDF contain the highest percentage of LDaQ (99,9% and 96.7%, respectively), however we also notice that the number of unique patterns in these two datasets is very small (only 5 for BM and 14 for BIO2RDF) which means that, possibly, the queries in these collections come from fixed templates. As regards DBPEDIA, the largest and most popular dataset in our collection, we see that the majority of its unique queries (87.7%) are potentially LDaQ.

Figures 13-15 show the distribution of LDaQ and non-LDaQ for LGD, SWDF, and DBPEDIA. We notice that all follow a similar power-law distribution: there is a very small number of patterns having a very large number of queries and a long tail of patterns each one having only a few queries. The top-10 LDaQ patterns in LGD and SWDF correspond to the 95% of all answerable queries, and the top-10 non-LDaQ to the 98% and 96%, respectively, of all non-answerable queries. Regarding the DBPEDIA dataset, the top-10 LDaQ patterns correspond to the 84% of the answerable queries and the non-LDaQ to the 86% of the non-answerable queries.

The listings in Figures 16-18 show the top-5 LDaQ and non-LDaQ patterns for LGD, SWDF and DBPEDIA (where `UN` = UNION, `OPT` = OPTIONAL). We notice that the majority of the frequent LDaQ patterns are short and request either the properties of a URI (like the patterns `[U] [V]`

[7]The implementation of pattern extraction is publicly available (cf. Footnote 3).

Table 1: Dataset statistics.

Dataset	#Queries	#Invalid	#Unconsidered	#Remaining	#Unique
LGD	4,240,736	456,393	1,148,809	2,635,534	670,809
SWDF	13,990,138	224,849	3,326,767	10,438,522	789,049
BM	129,989	0	0	129,989	129,989
BIO2RDF	192,057	47	2	192,008	62,819
DBPEDIA	49,296,201	2,003,381	3,869,723	43,423,097	16,028,271

Table 2: Linked Data answerable and no-answerable queries and unique patterns.

Dataset	#Test queries	#LDaQ	#LDaQ patterns	#Non-LDaQ	#non-LDaQ patterns
LGD	670,809	572,720 (85.4%)	444	98,089 (14.6%)	197
SWDF	789,049	677,923 (85.9%)	570	111,126 (14.1%)	202
BM	129,989	129,936 (99.9%)	4	53 (0.04%)	1
BIO2RDF	62,819	60,740 (96.7%)	9	2,079 (3.30%)	5
DBPEDIA	16,028,271	14,053,584 (87.7%)	2,816	1,974,687 (12.3%)	780

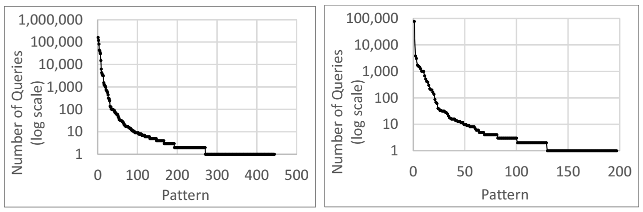

Figure 13: LDaQ (left) and non-LDaQ (right) pattern distribution in LGD.

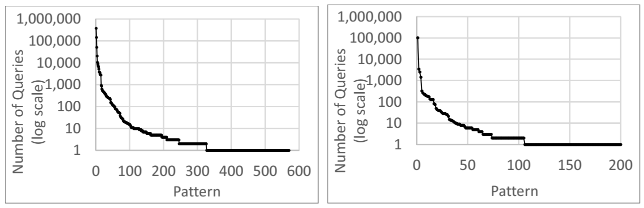

Figure 14: LDaQ (left) and non-LDaQ (right) pattern distribution in SWDF.

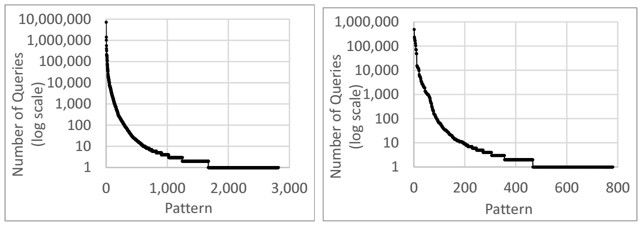

Figure 15: LDaQ (left) and non-LDaQ (right) pattern distribution in DBPEDIA.

`[V]` and `[U] [U] [V]`) or the URIs having a specific property value (like the patterns `[V] [V] [U]` and `[V] [U] [U]`). Regarding the non-LDaQ queries, we see that `[V] [U] [V]` and `[V] [U] [L]` are the more frequent patterns. In DBPEDIA, it is interesting that some of the top patterns are long and contain many UNION and OPTIONAL operators. These patterns probably correspond to a large number of similar (template-based) queries, possibly submitted by a small number of clients.

The full lists of LDaQ and non-LDaQ patterns are publicly available (cf. Footnote 3).

6.3 Efficiency of answerable queries

6.3.1 Querying a single URI

This is the simplest case where we request one or more properties (incoming or outgoing) of a single resource (patterns like `[U] [V] [V]` and `[V] [U] [U]`). This query type corresponds to around 77% of all unique queries in the SWDF dataset, 70% in LGD, 97% in BIO2RDF, and 56% in DBpedia.

As shown in [8], the time to answer this query type is proportional to the number of triples contained in the resource. Querying a resource of 10,000 triples requires around 1 sec while the time increases to 30 secs for resources with 1M triples. The same work examined the case of querying DBpedia URIs and showed that the average query time is around 320 ms if we access the N3 files and 650 ms through content negotiation, while the time to run the same query at DBpedia's endpoint is around 300 ms. Requesting one or more of the outgoing properties of a URI corresponds to 52% of all unique queries in the examined DBPEDIA dataset. This means that more than half of the queries can bypass the endpoint and be efficiently answered through link traversal. In general, this query type does not increase the data that is transferred over the network, while for queries requesting the outgoing properties of a specific entity, the query execution time is very low (since the number of triples is usually small). The time can be high for queries request-

```
1 [U] [V] [V]
2 [V] [V] [U]
3 OPT { [U] [U] [V] }
4 [V] [U] [U] ; [V] [V]
5 [V] [U] [U]
```

```
1 [V] [U] [V]
2 [V] [U] [V] . [V] [U] [V]
3 [V] [U] [L]
4 [V] [U] [L] OPT {[V] [U] [V]} OPT {[V] [U] [V]}
5 [V] [V] [V]
```

Figure 16: Top-5 LDaQ (up) and non-LDaQ (down) patterns in LGD.

```
1 [U] [U] [V]
2 [V] [U] [U]
3 [U] [V] [V]
4 { [U] [V] [V] } UN { [V] [V] [U] }
5 [U] [V] [V] OPT { [U] [U] [V] }
```

```
1 [V] [U] [L]
2 [V] [U] [V]
3 [V] [U] [L] ; [U] [V] . [V] [U] [V]
4 OPT { [V] [U] [V] }
5 [V] [U] [V] ; [U] [V]
```

Figure 17: Top-5 LDaQ (up) and non-LDaQ (down) patterns in SWDF.

```
1 [U] [U] [V]
2 { [U] [U] [U] } UN { [U] [U] [U] }
3 [V] [U] [U] ; [U] [L] . [V] [U] [U] { [V] [U] [V] } UN { [V]
  [U] [V] } UN { [V] [U] [V] } UN { [V] [U] [V] } { [V] [U]
  [V] } UN { [V] [U] [V] } OPT { [V] [U] [V] } OPT { [V] [U]
  [V] } OPT { [V] [U] [V] }
4 [U] [V] [V]
5 { [V] [U] [U] } UN { [V] [U] [U] } [V] [U] [L] . [V] [U] [V] ;
  [U] [L] ; [U] [V]
```

```
1 { [V] [U] [L] } UN { [U] [U] [V] } [V] [U] [V] ; [U] [V] .
  [V] [U] [V]
2 [V] [U] [L] ; [V] [V] OPT { [V] [U] [V] }
3 { [V] [U] [L] } UN { [V] [U] [V] ; [U] [L] } UN { [V] [U] [V] ;
  [U] [L] } OPT { [V] [U] [V] } OPT { [V] [U] [V] ; [U] [V] } OPT
  { [V] [U] [V] } OPT { [V] [U] [V] } OPT { [V] [U] [V] } OPT {
  [V] [U] [V] } OPT { [V] [U] [V] }
4 [V] [U] [L]
5 [V] [V] [V] . [V] [U] [L]
```

Figure 18: Top-5 LDaQ (up) and non-LDaQ (down) patterns in DBPEDIA.

ing the incoming properties of resources representing classes since in some cases the number of instances can be very large. For example, in DBpedia 2016, there are 3,218,716 instances of type *dbo:Person*.[8] If we consider that querying a resource of 1M triples requires around 30 seconds [8], the time to retrieve all instances of such a general class is around 1.5 minute. Requesting the incoming properties of a URI corresponds to around 3.6% of all unique queries in the DBPEDIA dataset, 20% in SWDF, and 25% in LGD.

6.3.2 Querying multiple URIs

This case includes the majority of queries containing joins (patterns like `[V] [U] [U] ; [U] [V]`). For instance, the query in Figure 5, which requests a specific property value (birth date) of all entities of a specific type (basketball players), is such a query. The query execution time in this case highly depends on the number of intermediate bindings.

[8]`http://wiki.dbpedia.org/dbpedia-2016-04-statistics`

Table 3: Query execution time (in sec) of the transformed SPARQL-LD queries for different number of intermediate bindings (resources to be fetched): (a) 136, (b) 262, (c) 502, (d) 1,030, (e) 9,787.

	(a)	(b)	(c)	(d)	(e)
Non-optimised	26	44	79	152	1,322
Optimised	7	13	24	48	423

We run experiments for the popular pattern `[V] [U] [U] ; [U] [V]` for different number of intermediate results. The submitted query requests the English label of all instances belonging to a particular class. We tested the following Wikicat classes containing varied number of instances: (a) American Civil Rights Lawyers (136 instances), (b) Video Artists (262 instances), (c) People From Sheffield (502 instances), (d) American Magazines (1,030 instances), and (e) American Male Film Actors (9,787 instances). We run the queries 10 times in different time points, and computed the average time to execute the corresponding SPARQL-LD query and store the results. We tried two different methods: i) non-optimised (sequential fetching of remote resources), and ii) optimised (using a simple parallelisation method which runs maximum 10 parallel threads at the same time for fetching the remote resources).

Table 3 shows the results. As expected, the query execution time is proportional to the number of intermediate bindings since the query needs to fetch the triples of each binding corresponding to a URI. We see that for large number of bindings the query execution time can be very high, especially if we do not optimise the query evaluation process. Such queries can highly increase the traffic of the HTTP server, thus the corresponding SPARQL-LD implementation should apply a politeness policy (c.f. Section 4.5).

7. CONCLUSIONS

We have investigated a SPARQL query evaluation method where the queries are directly answered on the live Web of Data through zero-knowledge link traversal, without accessing local or remote endpoints and without considering a starting graph or seed URIs. We introduced a method for checking if a SPARQL query can be answered through this query evaluation method and analysed a large dataset of real query logs for identifying frequent answerable and non-answerable query patterns. The analysis showed that more than 85% of the examined queries are potentially answerable, while the majority (>84%) of both answerable and non-answerable queries follow a few (≤10) specific patterns.

Subsequently we provided an algorithm for transforming answerable queries to SPARQL-LD queries that bypass the endpoints. Such a method to query Linked Data is based on standard and well-established Web technologies (HTTP, URI) and does not require the installation and maintenance of new servers and clients. With respect to the efficiency of the transformed queries, the query execution time highly depends on the number of remote resources that need to be accessed and the size of these resources (number of triples). We saw that more than half of the examined DBpedia queries can be answered through this method in < 1 sec. However, for queries with large number of intermediate bindings, which in turn might require large number of URI lookups, the query execution time can become prohibitively high. Finally, we studied the query characteristics that affect the cost of running an answerable query and provided an algorithm to approximate this cost. Approximating the query cost can help, for example, on deciding the query execution strategy to follow for answering the query.

As expected, the results showed that we cannot totally avoid the SPARQL endpoints and offer unrestricted query capabilities through this query execution method. We also expect that query evaluation is (almost) always faster in endpoints than through link traversal, since endpoints rely on pre-built indexes/databases. Nevertheless, our results showed that this query evaluation method can be offered efficiently for a large portion of queries, which could potentially decrease the load of these endpoints and increase their availability.

In future, we plan to evaluate different approaches on estimating the execution cost of answerable queries, from general ones that are widely applicable without requiring any prior knowledge (like the average number of object values for a given predicate) to more specific and accurate ones that make use of knowledge base statistics. Another interesting future direction is the design of adaptive query processing methods that combine different query execution strategies based on the load of the servers, the availability of the remote sources, and the estimated query execution cost. We also plan to study approaches on how to improve the execution time of the transformed SPARQL-LD queries, e.g., through caching or better query planning. Finally, further examination of the non-answerable query patterns is needed for understanding, for example, if a different policy for publishing Linked Data could be beneficial for making more queries answerable.

8. REFERENCES

[1] A. Bonifati, W. Martens, and T. Timm. An analytical study of large sparql query logs. *VLDB Endowment*, 11(2):149–161, 2017.

[2] A. Bonifati, W. Martens, and T. Timm. Darql: Deep analysis of sparql queries. In *Companion of the The Web Conference 2018*, pages 187–190. International World Wide Web Conferences Steering Committee, 2018.

[3] P. Bouquet, C. Ghidini, and L. Serafini. Querying the web of data: A formal approach. In *Asian Semantic Web Conference*, pages 291–305. Springer, 2009.

[4] C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 18(1):1–17, 2013.

[5] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *International Semantic Web Conference*, pages 277–293. Springer, 2013.

[6] P. Fafalios and Y. Tzitzikas. SPARQL-LD: A SPARQL Extension for Fetching and Querying Linked Data. In *The Semantic Web–ISWC 2015 (Posters & Demonstrations Track)*, Bethlehem, Pennsylvania, USA, 2015.

[7] P. Fafalios and Y. Tzitzikas. How Many and What Types of SPARQL Queries can be Answered through Zero-Knowledge Link Traversal? In *34th ACM/SIGAPP Symposium On Applied Computing*, Limassol, Cyprus, 2019.

[8] P. Fafalios, T. Yannakis, and Y. Tzitzikas. Querying the Web of Data with SPARQL-LD. In *International Conference on Theory and Practice of Digital Libraries*, pages 175–187. Springer, 2016.

[9] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres. Sparql 1.1 protocol. *Recommendation, W3C, March*, 2013.

[10] V. Fionda, G. Pirrò, and C. Gutierrez. NautiLOD: A formal language for the web of data graph. *ACM Transactions on the Web (TWEB)*, 9(1):5, 2015.

[11] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *19th international conference on World Wide Web*, pages 411–420. ACM, 2010.

[12] A. Harth and S. Speiser. On Completeness Classes for Query Evaluation on Linked Data. In *26th AAAI Conference on Artificial Intelligence*, 2012.

[13] O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *Extended Semantic Web Conference*, pages 154–169. Springer, 2011.

[14] O. Hartig. SPARQL for a Web of Linked Data: Semantics and computability. In *Extended Semantic Web Conference*, pages 8–23. Springer, 2012.

[15] O. Hartig. An overview on execution strategies for Linked Data queries. *Datenbank-Spektrum*, 13(2):89–99, 2013.

[16] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL queries over the web of linked data. In *International Semantic Web Conference*, pages 293–309. Springer, 2009.

[17] O. Hartig and J.-C. Freytag. Foundations of traversal based query execution over linked data. In *Proceedings of the 23rd ACM conference on Hypertext and social media*, pages 43–52. ACM, 2012.

[18] O. Hartig and J. Pérez. LDQL: A query language for the web of linked data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 41:9–29, 2016.

[19] T. Heath and C. Bizer. Linked Data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[20] A. Hogan, A. Harth, J. Umbrich, S. Kinsella, A. Polleres, and S. Decker. Searching and browsing linked data with swse: The semantic web search engine. *Web semantics: science, services and agents on the world wide web*, 9(4):365–401, 2011.

[21] A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *European Semantic Web Conference*, pages 493–507. Springer, 2008.

[22] M. Luczak-Roesch, S. Aljaloud, B. Berendt, L. Hollink, et al. Usewod 2016 research dataset (`http://usewod.org/`). 2016.

[23] D. P. Miranker, R. K. Depena, H. Jung, J. F. Sequeda, and C. Reyna. Diamond: A SPARQL query engine, for linked data based on the rete match. In *Workshop on Artificial Intelligence meets the Web of Data*, 2012.

[24] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems*, 34(3):16, 2009.

[25] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *European Semantic Web Conference*, pages 524–538. Springer, 2008.

[26] S. Sakr, M. Wylot, R. Mutharaju, D. Le Phuoc, and I. Fundulaki. *Centralized RDF Query Processing*, pages 33–49. Springer, 2018.

[27] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo. LSQ: the linked SPARQL queries dataset. In *International Semantic Web Conference*, pages 261–269. Springer, 2015.

[28] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web*, 7(5):493–518, 2016.

[29] Y. Tian, J. Umbrich, and Y. Yu. Enhancing source selection for live queries over linked data via query log mining. In *Joint International Semantic Technology Conference*, pages 176–191. Springer, 2011.

[30] Y. Tzitzikas, N. Minadakis, Y. Marketakis, P. Fafalios, C. Allocca, M. Mountantonakis, and I. Zidianaki. Matware: Constructing and exploiting domain specific warehouses by aggregating semantic data. In *11th Extended Semantic Web Conference (ESWC'14)*, May 2014.

[31] J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Link traversal querying for a diverse web of data. *Semantic Web*, 6(6):585–624, 2015.

[32] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5-6):495–544, 2011.

[33] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle. Querying datasets on the web with high availability. In *International Semantic Web Conference*, pages 180–196. Springer, 2014.

[34] R. Verborgh, T. Kuhn, and A. Sambra. Proceedings of the workshop on decentralizing the semantic web. In *Workshop on Decentralizing the Semantic Web, co-located with ISWC 2017*, 2017.

[35] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37:184–206, 2016.

[36] A. Wagner, T. T. Duc, G. Ladwig, A. Harth, and R. Studer. Top-k linked data query processing. In *Extended Semantic Web Conference*, pages 56–71. Springer, 2012.

[37] T. Yannakis, P. Fafalios, and Y. Tzitzikas. Heuristics-based query reordering for federated queries in sparql 1.1 and sparql-ld. In *2nd Workshop on Querying the Web of Data (QuWeDa'18)*, Heraklion, Greece, June 2018.

ABOUT THE AUTHORS:

Pavlos Fafalios is a postdoctoral researcher at the Information Systems Laboratory (ISL) and the Centre for Cultural Informatics (CCI) of FORTH-ICS (Greece) since June 2019. Before joining FORTH-ICS, he was a postdoctoral researcher at the L3S Research Center of the Leibniz University of Hanover (Germany). He obtained his MSc and PhD in 2012 and 2016, respectively, from the Computer Science Department of University of Crete (Greece). During his graduate studies, he was also a member of ISL at FORTH-ICS where he conducted basic and applied research within the context of several EU-founded research projects. His research interests fall in the areas of Information Retrieval, Semantic Web, Knowledge Representation, and Data Mining, with focus one exploratory and semantic search, semantic data management, and Linked Data. The results of his research have been published in more than 35 refereed research articles.

Yannis Tzitzikas is currently an associate professor in the Computer Science Department at the University of Crete (Greece) and Affiliated Researcher of the Information Systems Laboratory at FORTH-ICS (Greece). Before joining University of Crete and FORTH-ICS he was postdoctoral fellow at the University of Namur (Belgium) and ERCIM postdoctoral fellow at ISTI-CNR (Pisa, Italy) and at VTT Technical Research Centre of Finland. He conducted his undergraduate and graduate studies (MSc, PhD) in the Computer Science Department at the University of Crete. His research focuses on: exploratory search (principles, techniques, modern applications such as conversational faceted search), semantic data management (comparison functions, knowledge evolution, indexes, integration, visualization), and methodologies and technologies for building advanced information systems for digital preservation. The results of his research have been published in more than 140 papers in refereed international conferences and journals.