How Many and What Types of SPARQL Queries can be Answered through Zero-Knowledge Link Traversal?

Pavlos Fafalios
L3S Research Center, Leibniz University of Hanover
Hannover, Germany
fafalios@L3S.de

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

ABSTRACT

The current de-facto way to query the Web of Data is through the SPARQL protocol, where a client sends queries to a server through a SPARQL endpoint. Contrary to an HTTP server, providing and maintaining a robust and reliable endpoint requires a significant effort that not all publishers are willing or able to make. An alternative query evaluation method is through link traversal, where a query is answered by dereferencing online web resources (URIs) at real 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 live Web, without accessing local or remote endpoints and without a-priori knowledge of available data sources. In this paper, we first provide a method for checking if a SPARQL query (to be evaluated on a SPARQL endpoint) can be answered through zero-knowledge link traversal (without accessing the endpoint), 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. 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; Linked Data; Web of Data; SPARQL-LD

ACM Reference Format:
Pavlos Fafalios and Yannis Tzitzikas. 2019. How Many and What Types of SPARQL Queries can be Answered through Zero-Knowledge Link Traversal?. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SAC '19, April 8–12, 2019, Limassol, Cyprus
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Linked Data principles [18] has enabled the extension of the Web with a global data space based on open standards and protocols (the so-called *Web of Data*). 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 [8].

However, the low reliability of SPARQL endpoints is the major bottleneck that deters the exploitation of these knowledge bases by real applications [5, 32]. Publicly available endpoints are not optimised for efficiency and they often do not serve many concurrent requests in order to avoid server overloading. For instance, [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 a difficult challenge. 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). Thus, there arises the need of alternative, less demanding methods to query Web data [16, 32].

Link traversal, in particular, is a query processing method which relies on the Linked Data principles to answer a query by dereferencing online web resources (URIs) dynamically (at query execution time) [14, 16]. Inspired by this line of research, in this paper we study 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 also 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 first provide a method for checking if a query (to be evaluated on a SPARQL endpoint) can be answered without accessing any endpoint. We call this query type *Linked Data-answerable Queries (LDaQ)*. Then, we analyze a large corpus of real query logs from known SPARQL endpoints and study the patterns and frequency of both LDaQ and non-LDaQ. We find that more

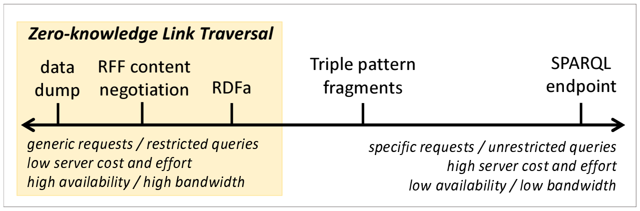

Zero-knowledge Link Traversal
data dump
RFF content negotiation
RDFa
Triple pattern fragments
SPARQL endpoint
generic requests / restricted queries
low server cost and effort
high availability / high bandwidth
specific requests / unrestricted queries
high server cost and effort
low availability / low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

Figure 1: Interfaces that allow querying Web data and positioning of zero-knowledge link traversal (this figure is a variation of the figure in [30]).

than 85% of the examined queries are potentially LDaQ, while the majority of them (>84%) follow a few patterns (≤10). Then, by exploiting SPARQL-LD [6], a SPARQL 1.1 extension that enables querying any HTTP resource containing RDF data, we provide an algorithm for transforming LDaQ to SPARQL-LD queries that bypass the endpoints. 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 < 1 sec, however for queries with large number of intermediate bindings the query execution time can become prohibitively high, thus calling for optimisation methods.

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.[1]

The rest of this paper is organised as follows: Section 2 presents the related literature. Section 3 motivates our work and describes the problem. Section 4 introduces the methods for finding LDaQ and transforming them to SPARQL-LD queries. Section 5 presents experimental results. Finally, Section 6 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 RDF data gathered from different sources [24, 28]. The current de-facto way for querying such repositories is through SPARQL. 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 [30, 32] where the authors introduced *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 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 [23] and SemWIQ [19] 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. [26] 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. Similar to data centralisation, query federation requires the data to be available through SPARQL endpoints.

SPARQL-LD [6, 7] 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. 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. [34] 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 [18] to dynamically discover data relevant for answering a query [14]. The work in [13, 15] 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 [21] is a similar in spirit query engine to evaluate SPARQL queries through link traversal. [12] 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], [13] and [11] discuss the notion of *completeness* and propose semantics to restrict the range of link traversal queries. Finally, index-based approaches rely on pre-built indexes for finding sources to look up during query execution [10, 27, 33].

Regarding more recent works, [29] studies the effectiveness of link traversal-based query execution and proposes reasoning extensions to help finding additional answers. [9] introduces a declarative navigational language for Linked Data, called NautiLOD, which enables to specify data sources by combining navigation and querying capabilities. Finally, LDQL [17] is a more expressive declarative language to query Linked Data through link traversal.

In our work, we focus on zero-knowledge link traversal and study the types of queries that can by directly answered on the live Web of Data by looking up URIs. The starting point of link traversal is *only* the URI(s) that exist in the query's graph pattern and additional

[1] https://github.com/fafalios/LDaQ

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 corresponds to the *query-reachable* completeness class as introduced in [11].

To our knowledge, our work is the first that i) 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, and ii) provides open source methods to detect answerable queries and transform them to SPARQL-LD queries that are evaluated without accessing endpoints or indexes. While recent works have conducted extensive analytical studies on the syntactical and structural characteristics of real SPARQL queries [1, 2, 25], no previous work has analysed queries in terms of their answerability through link traversal.

3 MOTIVATION & PROBLEM DESCRIPTION

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:

- 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.
- The capability to easily run queries directly on the Web of Data, from any client that supports SPARQL, without the need to build and maintain indexes and without requiring data providers to setup and maintain costly servers.
- To encourage decentralisation: the Web of Data is increasingly becoming a centralised data space relying on server-side infrastructures [31]. 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.

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.

```
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 <http://dbpedia.org/resource/Barack_Obama> {
    dbr:Barack_Obama dbo:birthDate ?birthDate } }
```

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

```
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 <http://dbpedia.org/ontology/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.

However, not all queries can be transformed to SPARQL-LD queries. Figure 6 shows two such queries. The left query requests all things (of unknown type) having the name *"Michael Jordan"*, while the right requests the total number of triples. Notice that the left 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`).

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

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

Figure 6: Example of non-LDaQ.

We now define two simple 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 (all triples where the URI is the *subject* or *object*). 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 [18]. 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 5.3.

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 problems and limitations (Section 4.4).

The implementation of all algorithms described in this section is publicly available as open source (see Footnote 1).

4.1 Linked Data-answerable Graph Patterns and Queries

Following the definitions of [22], 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 4.1. A BGP is Linked Data-answerable, for short LD-aBGP, 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.: $\forall t_i \in T,\ s(t_i) \in \mathcal{U}\ \vee\ o(t_i) \in \mathcal{U}\ \vee\ s(t_i) \in \mathcal{V}_i^b\ \vee\ o(t_i) \in \mathcal{V}_i^b$.

This definition corresponds to the *query-reachable* completeness class and the *completely-answerable* BGPs as introduced in [11].

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 4.2. 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 4.3. 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
$LB = \{\}$                                              ▷ Locally-bound variables
$M = \{\}$                    ▷ Map a variable to other variables that can help binding it
if B != null then $LB.addAll(B)$
for $t : triples(P)$ do
    if isURI(t.subject) then
        if isVariable(t.object) then LB.add(t.object)          ▷ Object variable can be bound
        if isVariable(t.predicate) then LB.add(t.predicate)    ▷ Predicate variable can be bound
    else if isURI(t.object) then
        if isVariable(t.subject) then LB.add(t.subject)        ▷ Subject variable can be bound
        if isVariable(t.predicate) then LB.add(t.predicate)    ▷ Predicate variable can be bound
    else if isVariable(t.subject) & isVariable(t.object) then
        if (LB.contains(t.subject) then LB.add(T.object)
            if (isVariable(t.predicate) then LB.add(T.predicate)
        else if (LB.contains(t.object) then LB.add(T.subject)
            if (isVariable(t.predicate) then LB.add(T.predicate)
        else
            M.add(t.subject, t.object)   ▷ Binding of object variable can bind the subject variable
            M.add(t.object, t.subject)   ▷ Binding of subject variable can bind the object variable
            if isVariable(t.predicate) then M.add(t.predicate, {t.subject, t.object})
$V$ = getAllVariables($P$)                     ▷ Set containing all graph pattern variables
for $v : V$ do                                ▷ Check for any variable that cannot be bound
    if $v \notin LB$ then
        if $!isBindable(M.get(v))$ then return false      ▷ Recursively check if the variable
  can be bound through the bindings of other variables
if $!inUnion$ then $B.addAll(LB)$;                   ▷ Add to $B$ the locally bound variables
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 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 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 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 LDaQ offers the ability to directly make use of this query execution method through existing

Algorithm 2 isLDaQ

```
Input: $Q$: Query graph pattern
Output: true or false
$B = \{\}; PENDING = \{\}$                ▷ Bound variables; Pending query elements
$E$ = getElements($Q$)    ▷ Each triple and each UNION group is considered a different element
for $e : E$ do                                       ▷ For each triple or UNION group
    if $isTripleElement(e)$ then                                ▷ The element is a triple
        if $!isLDaBGP(e, false, B)$ then
            $PENDING.add(e)$              ▷ Add this element to the list of pending elements
    else                                                 ▷ The element is a UNION group
        $allAnswerable = true$
        for $u : getUnionGraphPatterns(e)$ do
            if $!isLDaBGP(u, true, B)$ then $allAnswerable = false$; break
        if $allAnswerable$ then $B.addAll(getVariables(e))$
        else $PENDING.add(e)$             ▷ Add this element to the list of pending elements
while $!PENDING.isEmpty()$ do                        ▷ While there exist pending elements
    $foundNew = false$           ▷ In each FOR loop we must find a new answerable element
    for $pe : PENDING$ do                                      ▷ For each pending element
        if $isTripleElement(pe)$ then
            if $isLDaBGP(pe, false, B)$ then
                $foundNew = true; PENDING.remove(pe)$
        else                                     ▷ The pending element is a UNION group
            $allAnswerable = true$
            for $u : getUnionGraphPatterns(pe)$ do
                if $!isLDaBGP(u, true, B)$ then $allAnswerable = false$; break
            if $allAnswerable$ then
                $foundNew = true; PENDING.remove(pe)$
                $B.addAll(getVariables(pe))$
    if $!foundNew$ then return $false$                 ▷ No new answerable element was found
return true                                          ▷ The query is Linked Data-answerable
```

Algorithm 3 transformBGP

```
Input: Basic graph pattern $P$
Output: SPARQL-LD query pattern $P'$
$P' = \{\}$                                                  ▷ SPARQL-LD graph pattern
$B = \{\}; PENDING = \{\}$                ▷ Bound variables; Pending triple patterns
for $t \in getTriples(P)$ do                                 ▷ For each triple pattern
    if $isURI(t.subject) \mid\mid isURI(t.object)$ then
        $u = isURI(t.subject) ? t.subject : t.object$ ▷ Consider subject or object URI
        if $P'.containsService(u)$ then       ▷ There is a service pattern for the same URI
            $P'.getService(u).add(t)$          ▷ Add the triple pattern to its graph pattern
        else $P'.add(newServicePattern(u, t))$            ▷ Create a new service pattern
        $updateBoundVariables(t, B)$                ▷ Update the set of bound variables
    else if $B.contains(t.subject) \mid\mid B.contains(t.object)$ then
        $v = B.contains(t.subject) ? t.subject\ t.object$     ▷ Consider the subject or
object variable
        if $P'.containsService(v)$ then  ▷ There is a service pattern for the same variable
            $P'.getService(v).add(t)$          ▷ Add the triple pattern to its graph pattern
        else $P'.add(newServicePattern(v, t))$            ▷ Create a new service pattern
        $updateBoundVariables(t, B)$
    else $PENDING.add(t)$                          ▷ Transform this triple pattern later
while $!PENDING.isEmpty()$ do
    for $pt \in PENDING$ do                              ▷ For each pending triple pattern
        if $B.contains(pt.subject) \mid\mid B.contains(pt.object)$ then
            $v = B.contains(pt.subject) ? t.subject\ t.object$
            if $P'.containsService(v)$ then $P'.getService(v).add(pt)$
            else $P'.add(newServicePattern(v, pt))$
            $updateBoundVariables(pt, B); PENDING.remove(pt)$
return $P'$
```

Algorithm 4 transformQuery

```
Input: $Q$: query graph pattern
Output: $Q'$: SPARQL-LD query pattern
$Q' = \{\}; B = \{\}; PENDING = \{\}$  ▷ SPARQL-LD pattern; Bound vars; Pending elements
$E$ = getElements($Q$)    ▷ Each triple and each UNION group is considered a different element
for $e : E$ do                                       ▷ For each triple or UNION group
    if $isTripleElement(e)$ then                                ▷ The element is a triple
        if $isLDaBGP(e, false, B)$ then INCLUDE($Q'$, $e$, $B$)
        else $PENDING.add(e)$                                       ▷ Transform it later
    else                                                 ▷ The element is a UNION group
        $allAnswerable = true$
        for $u : getUnionGraphPatterns(e)$ do
            if $!isLDaBGP(u, true, B)$ then $allAnswerable = false$; break
        if $allAnswerable$ then $B.addAll(getVariables(e))$; INCLUDE($Q'$, $e$, $B$)
        else $PENDING.add(e)$                                       ▷ Transform it later
while $!PENDING.isEmpty()$ do                        ▷ While there exist pending elements
    for $pe : PENDING$ do                                      ▷ For each pending element
        if $isTripleElement(pe)$ then
            if $isLDaBGP(pe, false, B)$ then
                INCLUDE($Q'$, $e$, $B$); $PENDING.remove(pe)$
        else                                     ▷ The pending element is a UNION group
            $allAnswerable = true$
            for $u : getUnionGraphPatterns(pe)$ do
                if $!isLDaBGP(u, true, B)$ then $allAnswerable = false$; break
            if $allAnswerable$ then
                INCLUDE($Q'$, $e$, $B$)
                $PENDING.remove(pe); B.addAll(getVariables(pe))$
return $Q'$
```

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, we decide to look-up only the subject URI. 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. 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.

4.4 Problems and Limitations

There are some data access issues that must be taken into account when running queries over the live Web of Data [14]. In brief, dereferencing a URI may result in the retrieval of an unforeseeable large set of RDF triples, while some servers put restrictions on clients such as serving only a limited number of requests per second. Thus, a link traversal-based query execution system should implement a politeness policy to avoid overloading servers, e.g., by respecting the *robots.txt* protocol that allows web sites to demand delays between subsequent requests from the same client.

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 checking its answerability). Finally, SERVICE patterns over remote endpoints can be also transformed to SPARQL-LD

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

queries if their graph pattern is Linked Data-answerable. We leave the implementation of all these cases as part of our future work.

5 EXPERIMENTAL RESULTS

5.1 Datasets

We experimented with real SPARQL query logs provided by the Linked SPARQL Queries Dataset (LSQ) [25] and the USEWOD series of workshops [20]. 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.

5.2 Pattern-based analysis of LDaQ and non-LDaQ

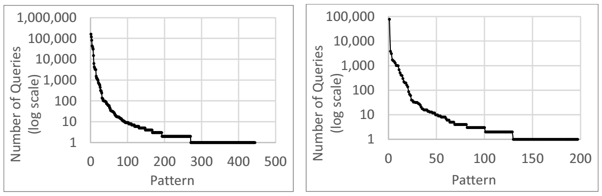
We examined the Linked Data-answerability of all unique queries (using Algorithm 2) as well as the pattern (template) they follow. For getting the pattern of a query, we considered only its graph pattern (text under WHERE), removed the FILTER operators, and replaced all variables, URIs, literals, and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

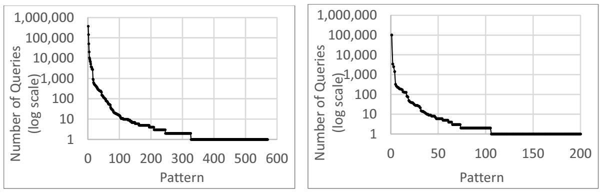
Table 2: Linked Data answerable and no-answerable queries and unique patterns.

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



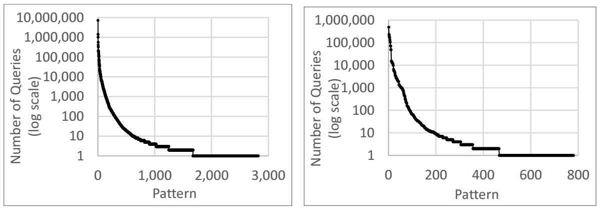
(a) Answerable patterns **(b) Not answerable patterns**

Figure 8: LDaQ and non-LDaQ pattern distribution in LGD.



(a) Answerable patterns **(b) Not answerable patterns**

Figure 9: LDaQ and non-LDaQ pattern distribution in SWDF.



(a) Answerable patterns **(b) Not answerable patterns**

Figure 10: LDaQ and non-LDaQ pattern distribution in DBPEDIA.

Figures 8-10 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 11-13 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] [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

[2] The implementation of pattern extraction is publicly available (cf. Footnote 1).

```
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 11: 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 12: 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 13: Top-5 LDaQ (up) and non-LDaQ (down) patterns in DBPEDIA.

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 1).

5.3 Efficiency of the transformed queries

5.3.1 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 [7], 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 requesting 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*.[3] If we consider that querying a resource of 1M triples requires around 30 seconds [7], 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.

Table 3: Query execution time (in seconds) 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

5.3.2 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.

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 4.4).

6 CONCLUSIONS

We essentially investigated the case where instead of having heavy-loaded servers (SPARQL endpoints) and light clients (SPARQL clients), we have very light servers (just Linked Data hosting) and heavier clients. This scenario could be beneficial not only in terms

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

of managerial costs, but also in terms of load balancing and robustness, however it could increase the data that should be transferred over the network and the overall query execution time.

To this end, we introduced a method for checking whether a SPARQL query can be answered on the live Web of Data without accessing any endpoint. We analysed a large dataset of real SPARQL query logs for identifying frequent answerable and non-answerable query patterns. The analysis showed that more than 85% of the examined queries are potentially Linked Data-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 Linked Data-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.

In general, we saw that, as expected, we cannot totally avoid SPARQL endpoints and offer unrestricted query capabilities through zero-knowledge link traversal. 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.

Regarding future work, an interesting 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 efficiency of query execution. Another interesting direction is the study of approaches to improve the execution time of the transformed SPARQL-LD queries, e.g., through caching or better query planning. Further examination of the non-answerable query patterns is also needed. For example, would a different policy for publishing Linked Data be beneficial for making more queries answerable?

REFERENCES

[1] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An analytical study of large SPARQL query logs. *VLDB Endowment* 11, 2 (2017), 149–161.

[2] Angela Bonifati, Wim Martens, and Thomas Timm. 2018. DARQL: Deep Analysis of SPARQL Queries. In *Companion of the The Web Conference 2018*. International World Wide Web Conferences Steering Committee, 187–190.

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

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

[5] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. 2013. SPARQL web-querying infrastructure: Ready for action?. In *International Semantic Web Conference*. Springer, 277–293.

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

[7] Pavlos Fafalios, Thanos Yannakis, and Yannis Tzitzikas. 2016. Querying the Web of Data with SPARQL-LD. In *International Conference on Theory and Practice of Digital Libraries*. Springer, 175–187.

[8] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. 2013. SPARQL 1.1 Protocol. *Recommendation, W3C, March* (2013).

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

[10] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. 2010. Data summaries for on-demand queries over linked data. In *19th international conference on World Wide Web*. ACM, 411–420.

[11] Andreas Harth and Sebastian Speiser. 2012. On Completeness Classes for Query Evaluation on Linked Data. In *26th AAAI Conference on Artificial Intelligence*.

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

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

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

[15] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. 2009. Executing SPARQL queries over the web of linked data. In *International Semantic Web Conference*. Springer, 293–309.

[16] Olaf Hartig and Johann-Christoph Freytag. 2012. Foundations of traversal based query execution over linked data. In *Proceedings of the 23rd ACM conference on Hypertext and social media*. ACM, 43–52.

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

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

[19] Andreas Langegger, Wolfram Wöß, and Martin Blöchl. 2008. A semantic web middleware for virtual data integration on the web. In *European Semantic Web Conference*. Springer, 493–507.

[20] Markus Luczak-Roesch, Saud Aljaloud, Bettina Berendt, Laura Hollink, et al. 2016. USEWOD 2016 Research Dataset (http://usewod.org/). (2016).

[21] Daniel P Miranker, Rodolfo K Depena, Hyunjoon Jung, Juan F Sequeda, and Carlos Reyna. 2012. Diamond: A SPARQL query engine, for linked data based on the rete match. In *Workshop on Artificial Intelligence meets the Web of Data*.

[22] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (2009), 16.

[23] Bastian Quilitz and Ulf Leser. 2008. Querying distributed RDF data sources with SPARQL. In *European Semantic Web Conference*. Springer, 524–538.

[24] Sherif Sakr, Marcin Wylot, Raghava Mutharaju, Danh Le Phuoc, and Irini Fundulaki. 2018. *Centralized RDF Query Processing*. Springer, 33–49.

[25] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. LSQ: the linked SPARQL queries dataset. In *International Semantic Web Conference*. Springer, 261–269.

[26] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. 2016. A fine-grained evaluation of SPARQL endpoint federation systems. *Semantic Web* 7, 5 (2016), 493–518.

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

[28] Y. Tzitzikas, N. Minadakis, Y. Marketakis, P. Fafalios, C. Allocca, M. Mountantonakis, and I. Zidianaki. 2014. MatWare: Constructing and Exploiting Domain Specific Warehouses by Aggregating Semantic Data. In *11th Extended Semantic Web Conference (ESWC'14)*.

[29] Jürgen Umbrich, Aidan Hogan, Axel Polleres, and Stefan Decker. 2015. Link traversal querying for a diverse web of data. *Semantic Web* 6, 6 (2015), 585–624.

[30] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. 2014. Querying datasets on the web with high availability. In *International Semantic Web Conference*. Springer, 180–196.

[31] Ruben Verborgh, Tobias Kuhn, and Andrei Sambra. 2017. Proceedings of the Workshop on Decentralizing the Semantic Web. In *Workshop on Decentralizing the Semantic Web, co-located with ISWC 2017*.

[32] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Web Semantics: Science, Services and Agents on the World Wide Web* 37 (2016), 184–206.

[33] Andreas Wagner, Thanh Tran Duc, Günter Ladwig, Andreas Harth, and Rudi Studer. 2012. Top-k linked data query processing. In *Extended Semantic Web Conference*. Springer, 56–71.

[34] T. Yannakis, P. Fafalios, and Y. Tzitzikas. 2018. 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.