Estimating the Cost of Executing Link Traversal based SPARQL Queries

Antonis Sklavos
antonis@sklavos.io
Information Systems Laboratory,
ICS-FORTH & University of Crete
Heraklion, Greece

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

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

ABSTRACT

An increasing number of organisations in almost all fields have started adopting semantic web technologies for publishing their data as open, linked and interoperable (RDF) datasets, queryable through the SPARQL language and protocol. *Link traversal* has emerged as a SPARQL query processing method that exploits the *Linked Data* principles and the dynamic nature of the Web to dynamically discover data relevant for answering a query by resolving online resources (URIs) during query evaluation. However, the execution time of link traversal queries can become prohibitively high for certain query types due to the high number of resources that need to be accessed during query execution. In this paper we propose and evaluate baseline methods for estimating the evaluation cost of link traversal queries. Such methods can be very useful for deciding on-the-fly the query execution strategy to follow for a given query, thereby reducing the load of a SPARQL endpoint and increasing the overall reliability of the query service. To evaluate the performance of the proposed methods, we have created (and make publicly available) a ground truth dataset consisting of 2,425 queries.

CCS CONCEPTS

• **Information systems** → *Query languages.*

KEYWORDS

Cost Estimation, Link Traversal, SPARQL, Linked Data, Web Data

1 INTRODUCTION

In the last years, a constantly increasing body of knowledge is made available on the web in the RDF format [21], including cross-domain knowledge bases, such as DBpedia [19] and Wikidata [29], but also domain-specific datasets, such as ClaimsKG [25] (fact-checking), ORKG [18] (scholarly communication), DrugBank [31, 32] (drugs), Semantic Layers [4] (web archives), Sampo portals [17] (digital humanities). In general, semantic technologies are increasingly used in a plethora of topical domains for making data available openly and interoperable for research and wide use [23]. Following the *Linked Data* principles[1] [15], such online RDF datasets can be directly accessed and queried by interested parties and external applications.

SPARQL is currently the standard language and protocol for querying and manipulating RDF datasets. However, the low reliability of SPARQL endpoints is the major bottleneck that deters the exploitation of these knowledge bases by real applications [2, 3]. For instance, [2] 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 up-times of 99-100%. The more recent work in [3] confirmed these performance and reliability issues, showing also that over a period of 6 months, at least 11% of the considered endpoints became less reliable.

Link traversal [13, 28] is an alternative SPARQL query processing method which relies on the *Linked Data* principles to answer a query by accessing (resolving) online web resources (URIs) dynamically, during query execution, without accessing endpoints. This query processing method is based on robust web protocols (HTTP, IRI), 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 servers/endpoints. However, the execution time of link traversal queries can become prohibitively high for certain types of queries due to the very high number of resources that need to be resolved at query execution time for retrieving their RDF triples [7]. This performance issue is a reason that deters the wider adoption of this query evaluation method.

In this paper, we focus on this problem and study methods to estimate the execution cost of queries that can be answered through link traversal. Our focus is, in particular, on *zero-knowledge* link traversal, which does not consider a starting graph or seed URIs for initiating the traversal, relying only on URIs that exist in the query pattern or that are dynamically retrieved during query execution.[2] We consider as *execution cost* the number of URIs that need to be accessed and resolved in real time because this affects both the query execution time and the amount of data that need to be transferred over the network, being at the same time independent of the underlying link traversal implementation/engine. By estimating this cost, a query service can decide on the fly the query execution strategy to follow for an incoming SPARQL query, based on factors such as the expected query execution time of link traversal and the availability (or current load) of the endpoint.

Fig. 1 shows a decision tree that can be considered by a SPARQL query service for deciding (in real time) on the query execution strategy to follow, aiming at improving the overall reliability of the query service without significantly affecting its response times. If the incoming query is answerable through zero-knowledge link traversal and the estimated execution cost is low, then the query will bypass the SPARQL endpoint (or the federation of the endpoints in case of a distributed environment) and will be executed through link traversal. If the cost is high, the service can check the availability of

[1] https://www.w3.org/DesignIssues/LinkedData.html

[2] According to [7], more than 85% of the queries submitted to five popular endpoints are answerable through zero-knowledge link traversal.

the endpoint(s), e.g., by running an ASK query. In case of availability, the query will be executed at the endpoint(s). Otherwise, the query will run though link traversal since it is preferable to get a delayed response than getting no results at all. Note here that the costs of checking the answerability of a query through link traversal, computing the link traversal cost, and checking the availability of a SPARQL endpoint, are negligible (a few ms). For endpoints that receive a high number of queries, such a query execution plan can highly reduce their load, and thus improve their overall reliability, since a large number of queries (of low link traversal cost) will bypass the endpoint(s) and be evaluated through link traversal using the robust HTTP protocol.

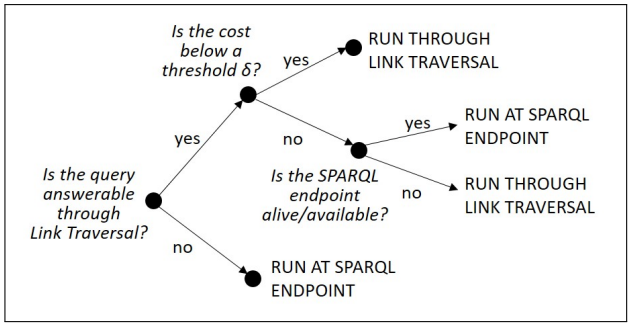

Figure 1: Deciding the query execution strategy to follow for answering a SPARQL query.

To enable the comparative evaluation of cost estimation methods, we build and make publicly available a ground truth dataset consisting of 2,425 queries. Using this ground truth, we experimentally evaluate four cost estimation methods that consider different aspects of the query, such as the type of predicates or the appearance of joins. The results showed that, considering predicate statistics, which can be computed easily (and only once) in a pre-processing step, together with joins and FILTER clauses, provides the best cost estimation performance.

In a nutshell, in this paper we make the following contributions:

- We study the main factors that affect the cost of executing a SPARQL query through zero-knowledge link traversal and propose a set of baseline methods to estimate it.
- We provide a ground truth dataset for the problem per se.
- We evaluate the performance of the baseline methods using the introduced ground truth dataset and a use case over a cross-domain knowledge base, in particular DBpedia [19].

The implementation of the proposed methods and the ground truth dataset are publicly available.[3]

The remainder of this paper is organised as follows: Sect. 2 provides the required background and describes related work. Sect. 3 details the main characteristics that affect the link traversal cost and introduces four baseline methods to estimate the cost. Sect. 4 introduces the ground truth dataset and presents evaluation results. Finally, Sect. 5 concludes the paper and discusses directions for future work.

[3]https://github.com/isl/LDAQ-CostEstimators

2 BACKGROUND AND RELATED WORK

2.1 Background

Link traversal is a SPARQL query execution method that accesses online resources in real time (during query evaluation) in order to retrieve the data needed for answering a SPARQL query [13, 28].

Consider, for instance, the below query that is to be executed over the DBpedia knowledge base:

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?birthDate  WHERE {
  dbr:Nelson_Mandela dbo:birthDate ?birthDate }
```

The query contains one triple pattern requesting the birth date of Nelson Mandela. If we run the query over the SPARQL endpoint of DBpedia, we get back the literal "1918-07-18" (xsd:date). To answer the query, a link traversal query execution method first needs to access the dereferenceable URI of Nelson Mandela (https://dbpedia.org/resource/Nelson_Mandela) for retrieving the RDF triples contained in this resource, and then evaluate the triple pattern over these RDF triples. In this example, only one resource needs to be accessed during query execution.[4]

Consider now the below query which requests all persons that influenced Plato together with a description of each of them:

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?influencer ?influencerDescription  WHERE {
  dbr:Plato dbo:influencedBy ?influencer .
  ?influencer dbo:abstract ?influencerDescription
        FILTER (lang(?influencerDescription) = 'en') }
```

In this case, link traversal first needs to access the URI of Plato in order to find the persons that influenced him, and then the URIs of all Plato's influencers for retrieving their description (14 influencers, according to DBpedia). Thus, link traversal needs to access 15 resources in total for answering this query.

A limitation of this query execution method is that not all queries are answerable through *zero-knowledge* link traversal. For example, the below query selects all triples whose object is a URI:

```
SELECT * WHERE {
  ?subject ?predicate ?object FILTER isURI(?object) }
```

This query cannot be answered without considering a starting graph or seed URIs, since there is no starting point (e.g., a URI in the query) that can be used for initiating the link traversal.

Our work focuses on queries that are answerable through zero-knowledge link traversal. The analysis of query logs in [7] has shown that such queries correspond to the majority (>85%) of the queries submitted to known endpoints.

An interesting way to directly run queries through zero-knowledge link traversal is by using SPARQL-LD [5, 8], 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 dereferencable URIs, online RDF files, or web pages embedded

[4]We consider that the URI of a predicate of a triple pattern is not dereferenced during link traversal because it (usually) does not contain all triples containing the particular URI as predicate, thus it does not help binding the subject or object variable.

with RDFa or JSON-LD.[5] The below query is an example of SPARQL-LD query that retrieves all persons that influenced Plato and their descriptions, without needing to access DBpedia's endpoint. The query first accesses the URI of Plato for retrieving his influencers (lines 4-5), and then queries the bound URI of each influencer for getting the description (lines 6-8):

```
1 PREFIX dbr: <http://dbpedia.org/resource/>
2 PREFIX dbo: <http://dbpedia.org/ontology/>
3 SELECT ?influencer ?influencerDescription  WHERE {
4   SERVICE dbr:Plato {
5      dbr:Plato dbo:influencedBy ?influencer }
6   SERVICE ?influencer {
7      ?influencer dbo:abstract ?influencerDescription
8            FILTER (lang(?influencerDescription) = 'en') } }
```

Since link traversal might need access to a large number of remote resources for answering a query (e.g., thousands of URIs), making query execution time prohibitively high, in this paper we provide methods that can estimate the query execution cost of a SPARQL query before its execution. Studying query execution strategies that exploit cost estimation (like the decision tree in Fig. 1) is out of the scope of this paper.

2.2 Related Work

We first review the related literature on link traversal and then position our work.

2.2.1 Link Traversal Approaches. Link Traversal exploits the *Linked Data* principles [15] to dynamically discover data relevant for answering a SPARQL query [28].

The approaches in [12, 14] and [22] follow RDF links by resolving URIs that exist 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. [11] 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 optimise query execution. [1], [12] and [10] discuss the notion of *completeness* and propose semantics to restrict the range of link traversal queries. [28] studies the effectiveness of link traversal-based query execution and proposes reasoning extensions to help finding additional answers.

Another direction of work on link traversal relies on pre-built indexes for finding sources to look up during query execution [9, 26, 30]. These approaches can determine all potentially relevant URIs at the beginning of query execution, which enables to fully parallelize the data retrieval process. However, there is a cost of initialising and maintaining the indexes.

The works in [7] and [6] focus on queries that can be answered directly on the live Web of Data, without considering a starting graph or seed URIs for initiating the link traversal. This *zero-knowledge* approach corresponds to the *query-reachable* completeness class as introduced in [10]. The starting point of link traversal in this case is one or more URIs that exist in the query's graph pattern while additional URIs are resolved only if this is needed for binding the variables of a triple pattern. The same works provide open source methods for i) examining the answerability of a query through zero-knowledge link traversal, and ii) transforming an answerable query to a SPARQL-LD query that is executed through this query evaluation method.[6]

2.2.2 Positioning. We focus on link traversal queries that can be answered directly on the live Web of Data, without considering a starting graph or seed URIs (zero-knowledge link traversal [6] and query-reachable completeness class [10]).

Our work extends [6] by (a) providing baseline methods to estimate the execution time of a query pattern that is answerable through zero-knowledge link traversal, (b) providing a ground truth dataset for the problem per se, and (c) evaluating the performance of the proposed methods using the introduced dataset and a use case over DBpedia.

Note here that, estimating the link traversal cost complements works on *query optimisation* [11, 16, 24, 27, 33], such as selectivity-based query reordering methods [11, 33]. Query optimisation tackles a related but different problem. For instance, a query executor can apply query pattern reordering before estimating the link traversal cost, or avoid applying optimisation if the estimated cost is low. Ideally, the query executor first applies query optimisation for defining the more efficient join order and then it computes the cost of zero-knowledge link traversal considering this fixed join order.

3 ESTIMATING THE QUERY COST

We consider as *query cost* the number of remote resources that have to be accessed and retrieved during query execution. This number affects both the query evaluation time as well as the amount of data that is transferred through the network, and is independent of the underlying implementation/engine of zero-knowledge link traversal.

Another factor that affects the query evaluation performance is the size of the remote resources, i.e., the number of triples contained in these resources. However, in a dynamic web context it is impossible to know in advance the number of triples contained in a remote resource without first accessing and retrieving it. Since we aim at estimating the cost of a query before its execution, this factor is not considered in our proposed methods.

Below, we first discuss the most common query characteristics that affect the link traversal cost (Sect. 3.1) and then introduce four methods to estimate the cost (Sect. 3.2).

3.1 Characteristics affecting the query cost

3.1.1 Characteristic #1. 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 are resolved during query evaluation for either binding variables or making the necessary joins.

3.1.2 Characteristic #2. 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.

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

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

Definition 3.1 (Necessary-to-resolve variable). A necessary-to-resolve variable in a query graph pattern is a variable whose URI bindings need to be resolved for binding another variable in the same query graph pattern.

Consider, for example, the below graph pattern which requests all authors together with the venues of their publications:

```
1 ?author a :Author .
2 ?author :hasPublication ?publication .
3 ?publication :inVenue ?venue
```

The pattern first has to access the URI of the `:Author` class for binding the variable `?author`. Then, it needs to access the URI of each author for binding the variable `?publication`. Thus, the variable `?author` is a necessary-to-resolve variable. Finally, the pattern needs to access the URI of each author publication for binding the variable `?venue`. Thus, `?publication` is another necessary to resolve variable, while the variable `?venue` is not since there is no need to resolve it.

If we arbitrarily consider that the number of authors returned by the `:Author` URI is 10,000 and that each author has 50 publications, then the *maximum* number of URIs that need to be accessed for evaluating the query is 510,001 (1 for binding `?author` + 10,000 for binding the variable `?publication` of each author + 10,000×50 for binding the variable `?venue` of each author publication).

Note here that there might be several common URIs in the bindings of one or more necessary-to-resolve variables (e.g., publications shared by multiple authors, in our example). Since it is impossible to know the values of the bindings without executing the query, we consider the *worst-case* scenario in which all bindings are different.

3.1.3 Characteristic #3. 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. For example, the below query pattern requests all URI properties of authors (i.e., their related entities) together with their labels:

```
1 ?author a :Author .
2 ?author ?property ?relatedEntity .
3 ?relatedEntity :label ?label
```

The number of bindings of the necessary-to-resolve variable `?relatedEntity` (which needs to be resolved for binding `?label`) can be very high since all properties of an author 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 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, we can expect a relatively large number of *object bindings* for the `:hasMember` predicate (depending on the context). 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 `:birthPlace` (there are many persons with the same birth place), but a small number for the predicate `:hasCapital` (there is only one country for a given capital city).

3.1.4 Characteristic #4. 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). Consider, for example, the below query:

```
1 ?author a :Author .
2 ?author :directorOf ?institution .
3 ?author :hasPublication ?publication .
4 ?publication :inVenue ?venue
```

The second triple pattern limits the bindings of the variable `?author` of the first triple pattern to only those having a `:directorOf` property (forming a star-shaped join). In this case, the number of bindings of the variable `?author` can be highly reduced before moving to the third pattern. For example, if from the 10,000 authors in the knowledge base only 100 are directors, then the maximum number of URIs that need to be accessed for evaluating the query is highly limited from 510,001 to 15,001 (1 for binding the variable `?author` + 10,000 for binding the variables `?institution` and `?publication` + 100×50 for binding the variable `?venue` of each author publication, if we roughly consider that each author has 50 publications).

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. For example, consider the below query pattern which instead of having the `:directorOf` property, it has a triple pattern (as first one) which requires the authors to belong to a specific party:

```
1 :party12 :hasMember ?author .
2 ?author a :Author .
3 ?author :hasPublication ?publication .
4 ?publication :inVenue ?venue
```

The two triples in lines 1 and 2 form a chain-shaped join which can highly limit the number of bindings of the variable `?author`.

3.1.5 Characteristic #5. Another query characteristic that can highly limit the bindings of a necessary-to-resolve variable and thus the query cost is the presence and position of FILTER clauses. The FILTER clause is a constraint which restricts the solutions (variable bindings) of the whole group of triple patterns in which it appears. Consider, for example, the below query pattern:

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

The pattern is the same with that of the first example of *Characteristic #4* 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 maximum 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).

3.1.6 Characteristic #6. A last characteristic that can potentially 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 query execution time if the underlying SPARQL implementation does not apply an optimisation technique, e.g., a query re-ordering method [33]. For instance, in the query pattern above (example of Characteristic #5), if we move the second and third triples to the end and the SPARQL implementation does not apply any pattern re-ordering method, the query cost is highly increased because the query needs to first retrieve the venues of all publications of all 10,000 authors before restricting the bindings of the ?author variable. For being widely applicable and implementation independent, we do not require that a specific query optimisation method must be applied before estimating the query cost.

3.2 Cost Estimation Methods

Considering the above-mentioned characteristics that can affect the query execution cost of link traversal, we now provide baseline methods to estimate it. The implementation of all methods is publicly available on GitHub (see Footnote 3).

3.2.1 Method 1 - Predicates agnostic ($M_{\neg p}$). Our first baseline method considers the number of URIs in the query (Characteristic #1) and also tries to estimate the expected number of bindings of each *necessary-to-resolve* variable (Characteristics #2 and #3). For the latter, we consider very limited knowledge about the underlying knowledge base. In particular, we consider constant values for the below five parameters:

(1) Average number of entity *outgoing* properties
(2) Average number of entity *incoming* properties
(3) Average number of *subject bindings* for any given property (except *rdf:type*) and object
(4) Average number of *subject bindings* for the property *rdf:type* and a given object/class (i.e., average number of instances per class)
(5) Average number of object bindings for a given subject and property

One can easily compute these values in a pre-processing step (and only once), e.g., by running SPARQL queries or accessing the RDF dumps. Listings 1-5 provide the queries for each of the five parameters.

3.2.2 Method 2 - Predicates aware (M_p). This method extends the first method by considering the actual value (URI) of each predicate in the query pattern (Characteristic #3). In particular, we can pre-compute (only once, in a pre-processing step) the *average number of subject bindings* (see query in Listing 6) and the *average number of object bindings* (see query in Listing. 7) for a large number of frequent predicates or for the full list of predicates used by the underlying knowledge base(s). For example, the predicate http://dbpedia.org/ontology/genre (the genre of a thing, e.g., of a music group, film, etc.) has average number of object bindings 1.8 (an object has, on average, around 2 genres) and average number of subject bindings 56.9 (there are, on average, around 57 objects having the same genre). These pre-computed numbers are then exploited in real time for estimating the number of bindings of the necessary-to-resolve variables contained in the query pattern. If the query pattern contains an unknown predicate, then we consider an average value as in $M_{\neg p}$.

Although the actual predicate selectivity can be skewed for a particular subject or object (since, for example, for the same predicate there might be subjects with very high number of object bindings and subjects with very low number), it provides a good estimate that is adequate in our (cost estimation) use case since it can distinguish the cases of always-small and always-large number of subject/object bindings.

3.2.3 Method 3 - Predicates+Joins aware (M_{pj}). Here we extend M_p by considering the star-shaped joins of necessary-to-resolve

```
SELECT (AVG(?count) AS ?average)
WHERE {
  { SELECT ?x (COUNT(DISTINCT ?y) AS ?count)
    WHERE {
      ?x a ?type . ?x ?y ?z } GROUP BY ?x }}
```

Listing 1: SPARQL query for computing the entities' average number of outgoing properties (Parameter #1).

```
SELECT (AVG(?count) AS ?average)
WHERE {
  { SELECT ?z (COUNT(DISTINCT ?y) AS ?count)
    WHERE {
      ?x ?y ?z . ?z a ?type } GROUP BY ?z }}
```

Listing 2: SPARQL query for computing the entities' average number of incoming properties (Parameter #2).

```
SELECT (AVG(?count) AS ?average)
WHERE {
 { SELECT ?z (COUNT(DISTINCT ?x) AS ?count)
   WHERE {
    ?x ?y ?z FILTER (?y!=rdf:type) } GROUP BY ?z } }
```

Listing 3: SPARQL query for computing the average number of subject bindings for any property except *rdf:type* (Parameter #3).

```
SELECT (AVG(?count) AS ?average)
WHERE {
  { SELECT ?z (COUNT(DISTINCT ?x) AS ?count)
    WHERE { ?x a ?z } GROUP BY ?z } }
```

Listing 4: SPARQL query for computing the average number of instances per class (Parameter #4).

```
SELECT (AVG(?count) AS ?average)
WHERE {
  { SELECT ?x (COUNT(DISTINCT ?z) AS ?count)
    WHERE { ?x ?y ?z } GROUP BY ?x } }
```

Listing 5: SPARQL query for computing the average number of object bindings for any property (Parameter #5).

```
SELECT (AVG(?count) AS ?average)
WHERE {
  { SELECT ?x (COUNT(DISTINCT ?z) AS ?count)
    WHERE { ?x <predicate> ?z } GROUP BY ?x } }
```

Listing 6: SPARQL query for computing the average number of object bindings for a particular predicate.

```
SELECT (AVG(?count) AS ?average)
WHERE {
  { SELECT ?z (COUNT(DISTINCT ?x) AS ?count)
    WHERE { ?x <predicate> ?z } GROUP BY ?z } }
```

Listing 7: SPARQL query for computing the average number of subject bindings for a particular predicate.

variables (Characteristic #4). In particular, if a necessary-to-resolve variable participates in a star-shaped join, we reduce its estimated number of bindings by a constant factor f_1 (which can take a value in the range [0.0, 1.0]). For instance, if $f_1 = 0.5$ and the estimated number of bindings of a necessary-to-resolve variable is 500, then this is reduced to 250 (0.5*500).

To decide on the value of f_1, we consider a set of training queries whose real cost is known (more in Sect. 4). Moreover, we do not consider the joins of the first and last query triple patterns since they do not affect the number of bindings of necessary-to-resolve variables (their bindings must be resolved regardless of whether they participate in star-shaped joins or not).

3.2.4 Method 4 - Predicates+Joins+Filters aware (M_{pjf}). Our last baseline method extends the previous method (M_{pj}) by also considering the appearance of FILTER clauses in the query graph pattern (Characteristic #5). In particular, similar to the case of joins, if a necessary-to-resolve variable exists in a FILTER clause and there is a triple pattern after that FILTER expression which requires resolving the URI bindings of the necessary-to-resolve variable, we reduce its estimated number of bindings by a constant factor f_2.

To decide on the value of f_2, we again consider a set of training queries for which we know their real cost (more in Sect. 4).

4 GROUND TRUTH DATASET & EVALUATION

We first describe the ground truth dataset we created for enabling the evaluation of cost estimation methods (Sect. 4.1). Then, we evaluate the four baseline methods described in the previous section (Sect. 4.2-4.3). Finally, we summarise the main findings (Sect. 4.4).

4.1 Ground Truth

The use of any cross-domain knowledge base (or federation of knowledge bases) providing resolvable URIs is adequate for the objective of our evaluation (examining the performance of cost estimation methods); we only need a diverse set of queries that are answerable through zero-knowledge link traversal.

To this end, we built a ground truth dataset by using *real* query logs of DBpedia provided by the USEWOD series of workshops [20]. In particular, we gathered a set of distinct queries and computed their real link traversal cost. To compute the cost, we first transformed them to SPARQL-LD [8] queries using the algorithm provided in [7] and without applying any pattern reordering method. Then, we executed the SPARQL-LD queries and counted the number of remote resources that each query needed to access for providing the results through zero-knowledge link traversal.[7]

To build the dataset, we discarded queries that are not answerable through zero-knowledge link traversal (using the algorithm provided in [7]), as well as a large number of queries having *cost = 1*, i.e., queries that require accessing a single URI that appears in the query pattern (by excluding them we can preform a more representative evaluation of the introduced methods). Also, we did not consider queries that make use of the `UNION` keyword or property paths (handing such cases is part of our future work), as well as queries with errors or that timed out.

The final dataset consists of 2,425 queries (see Footnote 3 for an access link). For each query we provide:

i) the transformed `SPARQL-LD` query that executes the query pattern through zero-knowledge link traversal
ii) the real query cost of link traversal
iii) all URIs that had to be accessed by the link traversal, together with the date and time we run the query
iv) the Notation3 (N3) files containing the triples of all the accessed URIs (57,138 unique files in total for all queries)

4.2 Evaluation Setup

We evaluated the four methods described in the previous section ($M_{\neg p}$, M_p, M_{pj}, M_{pjf}) using the introduced ground truth dataset. First, we split the dataset randomly into two equal parts and used the one part (*train*) for optimising the factors f_1 and f_2 of methods 3 and 4, respectively, and the other part (*test*) for evaluating performance on unseen queries. The considered value for both f_1 and f_2 is 0.9. For the parameters of $M_{\neg p}$, we considered the below constant values (decided by running SPARQL queries on DBpedia):

(1) Average number of entity *outgoing* properties = 25
(2) Average number of entity *incoming* properties = 5
(3) Average number of *subject bindings* for any given (no *rdf:type*) property and object = 1,505
(4) Average number of *subject bindings* for the property *rdf:type* and a given object/class (i.e., average number of instances per class) = 848
(5) Average number of object bindings for a given subject and property = 1.86

To measure the performance of each method over the set of test queries Q and the considered knowledge base K, we consider the *average absolute difference* (AvgAbsDiff) between the real cost and the estimated cost for all queries in Q. In particular:

$$\text{AvgAbsDiff} = \frac{\sum_{q \in Q} |\text{realCost}(q, K) - \text{estimatedCost}(q, K)|}{|Q|}$$

We also consider the *percentage difference* of the average real cost compared to the average estimated cost for all test queries (%AvgDiff). This will show us if, on average, the estimated cost is larger or smaller compared to the real cost (and how much). In both measures, smaller values means better results (closer to the real cost).

[7] Any implementation of zero-knowledge link traversal that does not apply a query optimisation technique is expected to provide the same real cost.

4.3 Evaluation Results

Table 1 shows the results for the four methods on the full test dataset. We notice that M_p, M_{pj} and M_{pjf} have a similar performance (from +46% to +54% of the real cost), highly outperforming $M_{\neg p}$ (predicates agnostic). The best performance is achieved by M_{pjf} which is predicates-aware and also considers joins and filters.

Table 1: Results on full test dataset.

Method	AvgAbsDiff	%AvgDiff
$M_{\neg p}$	523.9	+1,037%
M_p	100.9	+54%
M_{pj}	98.5	+49%
M_{pjf}	97.8	+46%

To better understand the performance of M_{pj} (Predicate+Joins aware), and its difference compared to M_p (Predicates aware), we now consider only queries from the test dataset that contain star-shaped joins (453 queries, in total). Table 2 shows the results. We now see that the performance improvement of M_{pj} compared to M_p is much higher (from 137% of real cost to 114%) compared to the results of Table 1, suggesting the positive effect of considering star-shaped joins in cost estimation.

Table 2: Results on test dataset considering only queries with star-shaped joins.

Method	AvgAbsDiff	%AvgDiff
$M_{\neg p}$	480.6	+1,477%
M_p	91.1	+137%
M_{pj}	84.6	+114%
M_{pjf}	84.5	+113%

We do the same considering only queries that contain both star-shaped joins and filters (436 queries, in total), in order to examine the performance of M_{pjf} (Predicates+Joins+Filters aware) compared to M_{pj} (Predicates+Joins aware). Table 3 shows the results. We now see that the performance of these two methods is almost the same, with M_{pjf} slightly outperforming M_{pj}. This very small difference is justified by the small number of queries in our test dataset that contain star-shaped joins but no FILTER (453−436 = 17 queries, in total), as well as by the fact that a high percentage of queries in out test dataset (94%) contain the filter clause at the very end of their pattern (thus, not affecting the bindings of a necessary-to-resolve variable).

Table 3: Results on test dataset considering only queries with star-shaped joins *and* filters.

Method	AvgAbsDiff	%AvgDiff
$M_{\neg p}$	429.0	+1,340%
M_p	76.2	+93%
M_{pj}	71.2	+75%
M_{pjf}	71.2	+74%

In general, we notice that the estimated cost is higher than the real cost in all cases. This is an expected result because, as described in Sect. 3, we consider the worst-case scenario in which all bindings of the necessary-to-resolve variables are different (it is impossible to know the values of the bindings without first executing the query).

4.4 Executive Summary

The evaluation results demonstrate that pre-computing information about the predicates used in a knowledge base can highly improve the cost estimation performance, as expected. As we have seen, it is straightforward to pre-compute this information (e.g., by running SPARQL queries or accessing RDF dumps), and is something that needs to be done only once.

We also noticed that considering joins and FILTER clauses through constant reduction factors does not highly improve cost estimation compared to the predicates-aware method, as one would probably expect. This is a good motivation for studying more fine-grained methods of exploiting joins and filters that do not consider constant factors, e.g., through training based on specific patterns.

The four described baseline methods are generic, i.e., they can work over any knowledge base or federation of knowledge bases. We only need to pre-compute the information required by the predicates aware methods, while if such information is not available for a predicate we can use a constant (average) value.

Finally, we have defined *cost* as the number of remote resources that need to be accessed and retrieved during query execution. Here, multiple such resources can be downloaded in parallel, or one can use a caching mechanism for frequent resources or a pattern reordering method, which means that the *real* query execution performance can be significantly improved. Optimising the query execution of link traversal is a different problem and out of the scope of this paper.

5 CONCLUSIONS

We have analyzed the main query characteristics that affect the cost of executing SPARQL queries through *zero-knowledge link traversal*, a query execution method that relies on robust web protocols (HTTP, IRI) and the dynamic nature of the web for answering a query by accessing online resources during query evaluation.

Based on these query characteristics, we introduced four baseline methods to estimate the link traversal cost. The first method considers very limited knowledge about the underlying knowledge base, the second considers predicate statistics (average number of subject/object bindings, computed once in a pre-processing step), the third method extends the second by also considering star-shaped joins, while the last method extends all previous methods by also considering FILTER clauses.

Such query cost estimation methods can be very useful for deciding on-the-fly the query execution strategy to follow, based on factors such as the availability and load of the SPARQL endpoint(s) at query time, as well as the estimated cost of each considered query execution method. The aim is to improve the overall reliability of the query service without significantly affecting its response times.

To measure the performance of the cost estimation methods, we have created (and make publicly available; cf. Footnote 3) a benchmark comprising DBpedia queries that are answerable through zero-knowledge link traversal. The experiments over this ground truth have shown that, considering predicate statistics together

with joins and FILTER clauses provides the best cost estimation (around 46% higher than the real cost, on average).

In the future, we plan to study additional, more fine-grained cost estimation methods, e.g., one that do not consider constant factors in case of joins and FILTERs. Another direction for future work includes the implementation of query execution strategies that exploit cost estimation for deciding on-the-fly on the query evaluation method to follow.

ACKNOWLEDGEMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 890861 (Project "ReKnow").

REFERENCES

[1] Paolo Bouquet, Chiara Ghidini, and Luciano Serafini. 2009. Querying the web of data: A formal approach. In *Asian Semantic Web Conference*. Springer, 291–305.
[2] 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.
[3] Jeremy Debattista, Christoph Lange, Sören Auer, and Dominic Cortis. 2018. Evaluating the quality of the LOD cloud: An empirical investigation. *Semantic Web* 9, 6 (2018), 859–901.
[4] Pavlos Fafalios, Helge Holzmann, Vaibhav Kasturia, and Wolfgang Nejdl. 2017. Building and querying semantic layers for web archives. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 1–10.
[5] 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.
[6] Pavlos Fafalios and Yannis Tzitzikas. 2019. Answering SPARQL queries on the web of data through zero-knowledge link traversal. *ACM SIGAPP Applied Computing Review* 19, 3 (2019), 18–32.
[7] Pavlos Fafalios and Yannis Tzitzikas. 2019. How many and what types of SPARQL queries can be answered through zero-knowledge link traversal?. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2267–2274.
[8] 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.
[9] 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.
[10] Andreas Harth and Sebastian Speiser. 2012. On Completeness Classes for Query Evaluation on Linked Data. In *26th AAAI Conference on Artificial Intelligence*.
[11] 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.
[12] Olaf Hartig. 2012. SPARQL for a Web of Linked Data: Semantics and computability. In *Extended Semantic Web Conference*. Springer, 8–23.
[13] Olaf Hartig. 2013. An overview on execution strategies for Linked Data queries. *Datenbank-Spektrum* 13, 2 (2013), 89–99.
[14] 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.
[15] 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.
[16] Hai Huang and Chengfei Liu. 2011. Estimating selectivity for joined RDF triple patterns. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 1435–1444.
[17] Eero Hyvönen. 2022. Digital humanities on the Semantic Web: Sampo model and portal series. *Semantic Web* Preprint (2022), 1–16.
[18] Mohamad Yaser Jaradeh, Allard Oelen, Kheir Eddine Farfar, Manuel Prinz, Jennifer D'Souza, Gábor Kismihók, Markus Stocker, and Sören Auer. 2019. Open research knowledge graph: next generation infrastructure for semantic scholarly knowledge. In *Proceedings of the 10th K-CAP*. 243–246.
[19] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic web* 6, 2 (2015), 167–195.
[20] Markus Luczak-Roesch, Saud Aljaloud, Bettina Berendt, Laura Hollink, et al. 2016. USEWOD 2016 Research Dataset (http://usewod.org/). (2016).
[21] Frank Manola, Eric Miller, Brian McBride, et al. 2004. RDF primer. *W3C recommendation* 10, 1-107 (2004), 6.
[22] 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*.
[23] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the linked data best practices in different topical domains. In *International Semantic Web Conference*. Springer, 245–260.
[24] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*. 595–604.
[25] Andon Tchechmedjiev, Pavlos Fafalios, Katarina Boland, Malo Gasquet, Matthäus Zloch, Benjamin Zapilko, Stefan Dietze, and Konstantin Todorov. 2019. ClaimsKG: A knowledge graph of fact-checked claims. In *ISWC'19*. Springer, 309–324.
[26] 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.
[27] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology*. 324–335.
[28] 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.
[29] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.
[30] Andreas Wagner, Thanh Tran Duc, Günter Ladwig, Andreas Harth, and Rudi Studer. 2012. Top-k linked data query processing. In *ESWC'12*. Springer, 56–71.
[31] David S Wishart, Yannick D Feunang, An C Guo, Elvis J Lo, Ana Marcu, Jason R Grant, Tanvir Sajed, Daniel Johnson, Carin Li, Zinat Sayeeda, et al. 2018. DrugBank 5.0: a major update to the DrugBank database for 2018. *Nucleic acids research* 46, D1 (2018), D1074–D1082.
[32] David S Wishart, Craig Knox, An Chi Guo, Dean Cheng, Savita Shrivastava, Dan Tzur, Bijaya Gautam, and Murtaza Hassanali. 2008. DrugBank: a knowledge base for drugs, drug actions and drug targets. *Nucleic acids research* 36 (2008), D901–D906.
[33] 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.