A Hybrid Approach to Linked Data Query Processing with Time Constraints

Steven Lynden
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
(AIST) Tsukuba, Japan
steven.lynden@aist.go.jp

Isao Kojima
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
(AIST) Tsukuba, Japan
isao.kojima@aist.go.jp

Akiyoshi Matono
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
(AIST) Tsukuba, Japan
a.matono@aist.go.jp

Akihito Nakamura
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
(AIST) Tsukuba, Japan
nakamura-akihito@aist.go.jp

Makoto Yui
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
(AIST) Tsukuba, Japan
m.yui@aist.go.jp

ABSTRACT

In addition to RDF data within documents published according to the Linked Data principles, SPARQL endpoints are also a potential source of a great deal of Linked Data. The execution of queries using languages such as SPARQL can use utilise both of these types of data sources. In this paper we present a hybrid approach to answering SPARQL queries that makes use of both link traversal-based and distributed query processing-based approaches in order to combine query answering over the Web of Linked Data and SPARQL endpoints respectively. The technique differs from existing work in that link traversal and endpoint queries take place in parallel without a static query plan. It is demonstrated how, using a set of heuristics and optimisation techniques, this can be effective when answering queries with time constraints (incomplete answers are acceptable in order to minimise execution time). An evaluation of the technique is presented using the FedBench Linked Data queries with query execution time limited to 10 seconds, with an analysis of answers that can be provided within this time limit.

1. INTRODUCTION

There are a number of issues that make querying Linked Data a challenging problem. Accessing diverse data sources spread around the Web introduces unpredictability in terms of response times and the amount of data that will be returned, making static query plan optimisation difficult. Answering non-trivial queries often requires more than a simple request/response-style interaction with a data source. Many pages from a single provider may need to be accessed in sequence to traverse the links to answer a query. As is the case with Web crawlers, in this scenario it is necessary to avoid sending too many requests to a single server within short space of time. SPARQL [13] endpoints provide query interfaces to public users, where query processing costs are met by the provider. As many SPARQL queries can be computationally expensive to execute, most public endpoints have some form of fair use policy to provide some relatively constant level of service to a potentially large number of users. For example, some query interfaces may limit the number of results they provide to queries, or generate a cost estimate at query compilation time and refuse to execute queries with a cost greater than a certain threshold.

A consequence of these characteristics of Linked Data query processing is that complete query answering for many queries in a timely manner is currently unrealisable. Some query processors, attempting to utilise a diverse set of distributed resources, may result in query execution times in the order of tens of minutes for some queries, and while there may be applications for such queries, in many cases providing some kind of approximate or incomplete answer within certain time constraints is more useful. In this paper we propose an approach that aims to provide relevant, fresh results in a timely manner while conceding that results are unlikely to be complete. The proposed approach can be categorised as a hybrid of two query processing techniques, link traversal [14] and distributed query processing. As static query execution

Copyright held by the author/owner(s)

LDOW2013 May 14, 2013, Rio de Janeiro, Brazil

plans are difficult to generate in advance, this step is avoided and this paper contributes a technique where the submission of SPARQL queries to endpoints takes place in parallel, with the aim of retrieving the data required to answer the query within given time constraints.

The problem of federated SPARQL querying has been widely recognised, with federated extensions to the SPARQL query language for SPARQL 1.1 [8] which support a construct for sending parts of a SPARQL query to specific SPARQL services. This is implemented by using a SERVICE keyword to merge distributed data, however this requires explicitly indicating which services are to be accessed and the specific parts of the query should be executed against them. In this paper we study the problem of automatically optimising the query without knowledge of this in advance across both the Web of Linked Data and SPARQL services.

The remainder of this paper is structured as follows. Section 2 introduces the background area and related work. Section 3 describes the design and implementation of our approach, followed by a performance evaluation used the FedBench [21] Linked Data queries in Section 4 and finally some concluding remarks in Section 5.

2. BACKGROUND

Guidelines for publishing Linked Data [16] propose best practices, based on the W3C set of RDF standards [17], for using the RDF data model to publish structured data on the Web, using RDF links to interlink data from different data sources. Querying Linked Data requires accessing distributed data held at different locations and in different formats. Issues related to scale, with respect to data distribution and volume, in addition to the frequency with which it is updated, require federated queries to access data sources directly rather than data warehouse-based solutions.

2.1 Accessing Linked Data

First and foremost, Linked Data forms what is often referred to as a “Web of Data”, consisting of dereferenceable URIs which are accessed to provide information about whatever the URI refers to using some standard format, such as RDF/XML [6], or RDFa [5]. In addition to accessing such data via HTTP, RDF databases also exist, which in many cases can be accessed using the SPARQL query language and its associated protocol [12]. Public SPARQL endpoints, such as DBpedia [2], provide a source of Linked Data, some of which may be also published according to Linked Data principles (dereferenceable URIs), and some of which may contain URIs that are not dereferenceable but still vital when answering certain SPARQL queries. Sindice [7] provides a SPARQL query interface over RDF data that has been indexed via crawling the Web of Data. Other SPARQL endpoints may provide access to data that is not available in other Web documents, resulting in a scenario where the data required to answer SPARQL queries may be found via either the SPARQL protocol, Web documents, or both.

2.2 Distributed SPARQL Query Processing

Distributed SPARQL query processors are able to accept a SPARQL query and answer the query by decomposing it into multiple sub-queries each of which are sent to a SPARQL endpoint. The results of the sub-queries are then combined to answer the original query. The challenges here are deciding on an initial query plan, including which SPARQL endpoints to access and in what order, and adapting to the unpredictable nature of SPARQL endpoints in terms of query response time, availability etc. Generating an efficient query plan is generally only possible if some statistics are available concerning the data contained within accessed endpoints, for example if the endpoint has a Vocabulary of Interlinked Datasets (VoID) [3] description associated with it. Another significant challenge is caused by the fact that endpoints are made available at the cost of the service provider, therefore restrictions on the types of queries that can be executed, and the results they can return, are commonplace. For example, the DBpedia endpoint's fair use policy at the time of writing is as follows:

- Query execution time is limited to 2 minutes.
- A rate limitation is placed on the number of connections that can be made per second.

Such restrictions result in greater unpredictability and complexity when optimising query execution plans over multiple SPARQL endpoints.

2.3 Link Traversal based Query Processing

Link traversal-based queries utilise the Web of Data to dereference URIs in the query, followed by recursively dereferencing URIs in the RDF data obtained. This may be enhanced, for example by using indexes to find related data, for example by storing `owl:sameAs` links to quickly expand the set of possible links that can be explored. Issues in link traversal based query processing include how to prioritise which links to dereference to find the data that will be useful in answering the query and how to determine when to terminate, as traversing a Web-scale set of links is not guaranteed to terminate in a reasonable amount of time. Issues related to Web crawling, i.e. avoiding multiple repeated requests to the same domain, are also pertinent.

2.4 Related work

Federated SPARQL query processing is currently an active area of research, with various emerging techniques proposed as solutions to the above problems, such as [11, 20, 18, 10]. Link traversal based query processors also exist such as SQUIN [9]. Indexing approaches also exist such as the aforementioned Sindice, which perhaps provides the most convenient way to query the Semantic Web so far. The amount of Semantic Web data is growing however, in particular with the adoption of formats such as RDFa, use of which is becoming increasingly widespread. This means that indexing may be a costly approach that eventually cannot scale. The Semantic Web is also becoming more dynamic, for example with ontologies such as Good Relations [4] and schema.org being used to represent data that is constantly being updated, meaning that indexes such as Sindice may struggle to provide an up-to-date reflection of the Semantic Web.

The hybrid approach to Linked Data query processing was first proposed in [15] and work inspired by this idea is start-

ing to appear. One such approach that uses both SPARQL endpoints and RDF documents to ensure the freshness of SPARQL query results is [22]. Similar to our approach, this approach combines accessing RDF documents and SPARQL endpoints, however this is achieved by executing two query engines in sequence to perform each step using a static query plan. This contrasts to our approach which aims to do both in parallel within fixed time constraints and very limited static query planning.

3. HYBRID LINKED DATA QUERY PROCESSING WITH TIME CONSTRAINTS

In this section an approach toward hybrid Linked Data query processing is presented, which aims to combine link traversal and distributed query processing on-the-fly in order to retrieve relevant data within a fixed time boundary and present best-effort results within that time-frame. The main characteristics of this approach are:

- Parallel query execution: accessing multiple SPARQL endpoints and dereferencing Linked Data URIs takes place in parallel. The aim is to retrieve data from as many sources as possible while avoiding simultaneous or very fast repeated requests to the same providers.
- Optimise communication costs: the bottleneck when querying a diverse set of distributed resources is retrieving the data rather than processing it once it has been retrieved.
- Execute for a fixed amount of time: instead of allowing execution time to continue until all results are obtained, or a fixed number of results are obtained, it is assumed that a best effort should be made within a fixed amount of time to provide a result.

The process involved is illustrated in Figure 1, which consists of the following steps:

1. The initial SPARQL query is parsed and compiled into a set of triple patterns and the query execution component is initialised. This query is subsequently referred to as the federated query; other SPARQL queries issued to SPARQL endpoints to contribute towards answering the federated query are referred to as subqueries.
2. A *local graph* component is initialised, which will store intermediate results, i.e. triples which have been found to match the set of triple patterns in the query.
3. Two components are executed, the *endpoint query manager* and the *active discovery manager*. The endpoint query manager sends queries to SPARQL endpoints and the active discovery manager dereferences URIs, analogous to a link traversal style of query answering.
4. After a time t, for which the query is scheduled to run, the endpoint query manager and active discovery manager are terminated and the local graph component is used to obtain the result of the federated query.

3.1 Query Compilation

During query compilation the federated SPARQL query is decomposed into a set of triple patterns $TP = tp_1, tp_2, ..., tp_i$, where there are i triple patterns in the SPARQL query. Any applicable FILTER predicates are associated with a triple pattern by the compiler as they are potentially pushed down to SPARQL endpoints or applied by the active discovery manager. The local store component is initialised to be an empty RDF graph to which RDF data retrieved from data sources that matches patterns in TP will be added. The local graph has the following properties:

- When a triple m is added to the local graph that matches a triple pattern tp_i, any variables in tp_i are associated with the relevant subject, predicate or object value in m, for example if tp_i contains a subject variable, var, then the subject value of the triple m is stored as one of the bindings of var (each variable can have multiple bindings). Bindings in the local graph are not necessarily variable bindings in the query result; they have the potential to be so depending on the final evaluation of the query.
- Triples containing blank nodes and duplicate triples are ignored.
- If the predicate is an `owl:sameAs` link and the subject and object are both URIs from the DBpedia namespace, the value is ignored due to the fact that this often leads to processing `sameAs` links between different languages, which leads to an explosion of the amount of data in the local store which may be relatively unimportant for answering the query in many scenarios. As we are aiming to process queries within fixed time constraints, the assumption is made that following such links is not desirable and leave for future work how to detect cases where multi-lingual query results would be desirable and how more sophisticated ways to rank the importance of `sameAs` links could be integrated or substituted within the system.

Subsequently the active discovery manager and endpoint query manager are started, passing a parameter t, equal to the number of seconds for which the query should be executed and the empty local graph to which both components will have access.

3.2 Active Discovery Manager

The active discovery manager creates a single thread for each pay-level-domain (PLD) encountered during query processing, starting with the initial set of URIs in the query and starting new threads as they are encountered, as shown in Algorithm 1.

Separate threads are created by the active discovery manager in order to dereference URIs within that domain only, without swamping the domain with requests. Each thread implements a policy that ensures a gap of at least 0.5 seconds takes place before subsequent HTTP requests to a domain. Each thread first dereferences any URIs in the query itself, and then chooses URIs by calling Algorithm 2 every 0.5 seconds until the query execution engine is terminated. The URI is dereferenced and triple patterns in the

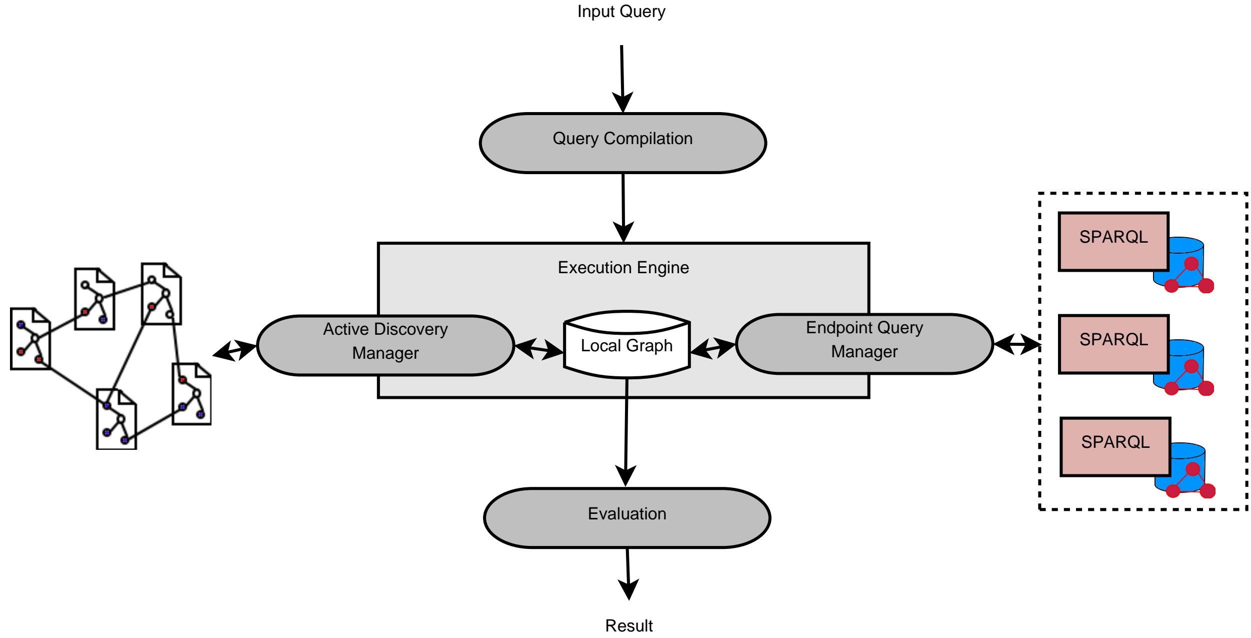

Figure 1: Hybrid approach

Once the query is compiled, two components are executed in parallel. The active discovery manager (left) aims to retrieve data from the Web of Linked Data. The endpoint query manager (right) aims to retrieve data from SPARQL endpoints. Linked data that matches triple patterns in the federated query are placed in the local graph, which is also utilised by the endpoint query manager and active discovery manager to retrieve new data. Following a fixed time, t, the query execution engine is terminated and the results extracted from the local graph.

Algorithm 1 The algorithm executed by the active discovery manager.

1: **for all** $tp_i \in TP$ **do**
2: $PLD := \emptyset$
3: **if** $tp_i.subject$ is a URI **then**
4: $PLD := PLD \cup getPLD(tp_i.subject)$
5: **end if**
6: **if** $tp_i.predicate$ is a URI **then**
7: $PLD := PLD \cup getPLD(tp_i.predicate)$
8: **end if**
9: **if** $tp_i.object$ is a URI **then**
10: $PLD := PLD \cup getPLD(tp_i.object)$
11: **end if**
12: **end for**
13: **while** $time < t$ **do**
14: $PLD := PLD \cup localGraph.getPLDs()$
15: **for all** $pld_i \in PLD$ **do**
16: Start **Active Discovery Thread for** pld_i **(if not yet started)**
17: **end for**
18: **end while**
19: **Terminate all active discovery threads**

Lines 1-12 add the PLDs from URIs in the query. The subsequent while loop starts new threads for these PLDs and for any PLDs encountered during query processing. $getPLD$ takes a URI and returns the PLD. $localGraph.getPLDs$ returns the set of PLDs from all URIs in the local graph.

query (plus any applicable FILTER predicates) are matched against the RDF data retrieved from the URI (if any), with matches added to the local graph. The ranking function, *bestRanking*, used by Algorithm 2, takes a set of URIs and aims to choose the one most likely to produce triples that match patterns in the query and therefore potentially yield results. As the number of URIs dereferenced by one active discovery thread will be relatively small (e.g. for a 10 second query a maximum of 20 requests assuming a 0.5 delay between requests), and nothing is known about the RDF data that can be retrieved from each URI before query execution, it is necessary to make a very quick and approximate estimation to choose one URI to dereference from a potentially large number of possible URIs. Although it is unrealistic to achieve significant improvements over a random selection with samples of less than 20, each active discovery thread selects a URI on the basis of string similarity, compared to URIs that have previously been dereferenced by the same thread (therefore within the same domain) and matched triples patterns. In order to see why, consider the following example based on FedBench Linked Data query 9:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dbc: <http://dbpedia.org/resource/category>
PREFIX dbp: <http://dbpedia.org/resource/property>
SELECT * WHERE {
   ?x dc:subject dbc:FIFA_World_Cup-winning_countries .
   ?p dbp:managerclubs ?x .
   ?p foaf:name "Luiz Felipe Scolari"@en .
}
```

For the http://dbpedia.org PLD, following the initial dereferencing of the URI in the first triple pattern, consider the following links that were dereferenced within that domain during a trial run of the execution of the active discovery manager (with the number of RDF triples retrieved that matched triple patterns in the query in brackets):

```
/resource/England_national_football_team (25)
/resource/Spain_national_football_team (23)
/resource/Brazil_national_team (18)
/resource/FC_Bunyodkor  (3)
/resource/Vicente_Feola  (2)
/resource/Luiz_Felipe_Scolari (16)
```

From this it seems intuitive that URIs containing strings similar to "national_football_team" and "Luiz_Felipe_Scolari" would be more likely to contain RDF triples that match query triple patterns. We base the selection of URIs on the assumption that the URI itself contains some clue about how many triple patterns will be matched (although they might not lead to actual query results, the active discovery manager aims to find as many triples as possible that *may* be used to produce results). Given a set of URIs from which a selection is to be made, the best *bestRanking* function used in Algorithm 2 ranks all URIs based on string distance measures between the URI and the set of dereferenced (visited) URIs, choosing a URI with the highest value obtained from the following function:

$$rank(uri) = \sum_{u \in U} [matched(u) * (1 - distance(u, uri))] \quad (1)$$

where the active discovery thread has visited the set of U URIs, and visiting URI $u \in U$ resulted in $matched(u)$ triples that matched triple patterns in the query. The function $distance(u, uri)$ gives the Levenshtein distance [19] between the two URIs represented as strings, normalised in the range [0-1].

Algorithm 2 The algorithm executed by each active discovery thread every 0.5 seconds until the query is terminated in order to choose a URI which should be dereferenced by the active discovery manager.

1: T := Set of URIs already dereferenced
2: $S1 := \emptyset$
3: $S2 := \emptyset$
4: **for all** $tp_i \in TP$ **do**
5:   **if** $localGraph.allVarsBound(tp_i)$ **then**
6:     $S2 := S2 \cup localGraph.getBindings(tp_i) - T$
7:   **else**
8:     $S1 := S1 \cup localGraph.getBindings(tp_i) - T$
9:   **end if**
10: **end for**
11: **if** $S1 \neq \emptyset$ **then**
12:   **return** bestRanking($S1$)
13: **else if** $S2 \neq \emptyset$ **then**
14:   **return** bestRanking($S2$)
15: **else**
16:   **return** null
17: **end if**

Any triples in RDF data retrieved from the URI are added to the local graph. $S1$ is a prioritised set of URIs that are more likely to match triples for which there are not yet bindings in the local graph (the $allVarsBound(tp)$ function supported by the local graph returns true if every variable in the triple pattern has at least one associated value). Prioritising $S1$ is done to ensure that at least some results will be provided by the query execution engine.

3.3 Endpoint Query Manager

The endpoint query manager utilises known SPARQL endpoints to retrieve triples that match query triple patterns and add them to the local graph. It is assumed prior to query execution that the system is configured with a set of endpoints to be used. Endpoint queries are performed by mapping each triple pattern to a data source, combining triple patterns into the same query where possible, and pushing down predicates where possible. To do this efficiently, some information is required about the RDF triples contained within each data source, which should be constructed during an initialisation/configuration step taking place before any queries over the data sources can be executed. The set of distinct predicate values is obtainable via a SPARQL SELECT query using the DISTINCT function to select the set of unique predicate values, or a sequence of SPARQL ASK queries incorporating specific predicate values (e.g. obtained from ontologies of interest or a specific query). This information enables a function $predMatch(tp_i, endpoint) \rightarrow [true|false]$ to be implemented which determines whether instances of a triple pattern can exist in an endpoint (the value will always be true if the triple pattern predicate is a variable).

The endpoint query manager creates a separate thread for each SPARQL endpoint to be used during query execution, where each thread sends SPARQL sub-queries to that endpoint in order to retrieve triples that will be used to answer the query. Each thread is designed to do the following:

- Execute simple queries by using the LIMIT feature of the SPARQL language and sending a small number of triple patterns. This provides results quickly, which can then be used by the active discovery manager to investigate URIs in parallel.
- Return relevant results by restricting the values of variables with FILTER clauses. For example, if the query contains a triple pattern $?v1$ <pred> $?v2$, sending this triple pattern alone in a sub-query to the endpoint would results in many bindings that are probably irrelevant in answering the query. Instead bindings from the local graph are added, for example assuming $v1$ has values '<u1>' and '<u2>', a sub-query of the following form may be constructed: `SELECT * WHERE { ?v1 <pred> ?v2 . FILTER (?v1=<u1>||?v1=<u2>) }`. This restricts sub-query results to those which are likely to be of use in answering the query.
- To avoid placing an excessive burden on a particular endpoint, the endpoint query manager enforces a 5 second gap between queries to the same endpoint. As the

aim of our approach is to execute queries within a timeframe of 10 seconds, this corresponds to each endpoint being queried twice at most.

Algorithm 3 The algorithm executed to compose sub-queries by each endpoint query manager thread.

```
1: sq :=new sub-query
2: if number of sub-queries sent= 0 then
3:    for all tp_i ∈ TP do
4:       if predMatch(tp_i, endpoint) then
5:          sq.addPattern(tp_i)
6:       end if
7:    end for
8: else
9:    for all var ∈ localGraph.boundVar() do
10:      sq1 :=new sub-query
11:      sq1.addBinds(localGraph.getBinds(var))
12:      for all tp_i ∈ TP do
13:         if predMatch(tp_i, endpoint) & tp_i.contains(var)
            then
14:            sq.addPattern(tp_i)
15:         end if
16:      end for
17:      if value(sq1) > value(sq) then
18:         sq := sq1
19:      end if
20:   end for
21: end if
22: if valid(sq) then
23:    return sq
24: else
25:    return null
26: end if
```

Line 2 ensures that for the generation of the first sub-query to the endpoint in question, all applicable triple patterns in the federated query are added to the sub-query and this query is returned (lines 3-7). Subsequent queries are constructed using bindings from the local graph to limit the results to those useful in answering the federated query. For each bound variable, *var*, the bindings for that variable are used to constrain the set of possible results by adding FILTER predicates to the sub-query. $localGraph.boundVar$ returns all bound variables in the local graph, and *getBinds* returns the values bound to a specific variable. A maximum of 10 values are used to avoid very large query strings and the *addBinds* operation also ignores any bindings that have already been sent/received by to/from the endpoint. Each triple pattern containing the variable *var* is then added to the potential sub query, $sq1$, which is compared to any previously generated sub-queries using a cost function (line 17) and the best sub-query finally returned, provided the constraints met by the *valid* function are satisfied.

These objectives are obtained by executing Algorithm 3, which selects a sub-query every 5 seconds until the query execution engine is terminated. Results from the queries are stored in the local graph shared by the active discovery manager.

When sending queries with bindings, the *value* function utilised by Algorithm 3 estimates the relative value of sending a sub-query *sq* to a given endpoint:

$$value(sq) = \sum_{var \in V} \frac{s}{max(1, localGraph.numBindings(var))} \quad (2)$$

where V is the set of non-bound variables in the sub-query, s is the number of different values in the FILTER clause used to bind the bound variable, and $localGraph.numBindings$ returns the number of bindings already stored in the local graph for a variable *var*. The function estimates the value of the sub-query by summing the fractional increase in variable bindings that would be obtained under the simplifying assumption that a new value is obtained for every bound variable sent. The motivation for this is that queries with the potential to bind variables with relatively few bindings in the local graph should be prioritised in order to potentially yield more results when the federated query is finally evaluated.

Algorithm 3 aims to select the best possible sub-query to send to an endpoint, however a query is only sent if the constraints of the *valid* function are met. *valid* returns true if:

1. The sub-query has bindings in the FILTER clause from the local graph. (Any sub-query with variable bindings is likely to return results that are useful with respect to obtaining a query result).

   OR

2. The sub-query contains at least two triple patterns which are joined by shared variables, or a single triple pattern with only one variable. (Single triple pattern sub-queries with more than one variable are unlikely to return relevant results).

3.4 Implementation

The proposed approach is implemented in Java, and the active discovery manager uses the standard Java libraries for retrieving documents via HTTP. Jena [1] is used by the endpoint query manager to send sub-queries to SPARQL endpoints and process the results. Jena is also used for RDF/XML parsing. The local graph component is also implemented using the Jena RDF graph model, and the Jena libraries are used once the query execution engine has finished to execute the federated query against the local graph to obtain the final query results.

4. EVALUATION

This section presents a performance analysis of the approach discussed in the previous section, made using the FedBench Linked Data queries. The aim of the evaluation is to analyse the extent to which results can be obtained under tight time constraints (10 seconds) when answering queries over the Web of Linked Data in conjunction with publicly available SPARQL endpoints. This section is structured as follows: firstly the FedBench Linked Data queries are described, secondly an analysis is presented of the results (while relaxing time constraints) that be obtained using (a) only link traversal and (b) sending the queries to specific SPARQL endpoints. Finally, a comparison is made with our proposed approach, using both link traversal and SPARQL endpoints in parallel under strict time constraints. The following SPARQL endpoints are utilised and accessed in parallel during query execution:

- Sindice:
  `http://sparql.sindice.com/sparql`
- DBpedia
  `http://dbpedia.org/sparql`
- Semanticweb.org
  `http://data.semanticweb.org/sparql`

Experiments are performed on a 2.4GHz Intel Xeon Dual Processor machine running Fedora 17 Linux with 16GB memory, connected to the Internet via a 100Mbs Ethernet LAN. The query execution engine is executed using a single Java Virtual Machine (version 1.7) using Jena version 2.7.1.

4.1 FedBench Linked Data Queries

FedBench is a benchmark suite for Semantic data query processing, aimed at allowing developers to test the performance of the query engines against a standard data set and analyse both efficiency and effectiveness. Various categories exist within FedBench, such as Life Science and cross domain queries, however in this paper we focus on the Linked Data queries, a set of basic graph pattern queries, designed primarily for link traversal style query processing. The queries are shown in Figure 2, along with some minor changes that were required to get some of them to return results. Two queries, 6 and 8, could not be executed with any combination of SPARQL endpoints and link traversal that we attempted (see Figure 2 for details) and are therefore omitted.

4.2 Using Link Traversal only

In order to determine a baseline for link traversal performance alone, the active discovery manager is executed without the endpoint query manager for 10 minutes. Table 1 shows the number of triples in the local graph and the number of query results. In all cases the time taken for a query result to be obtained from the local graph was negligible compared to query execution time (less than 40 milliseconds) Some queries ran out of URIs to dereference fairly quickly (queries 5 and 7), or halfway towards the 10 minute limit (query 10). Query 9 returned no results, showing that even if quite a lengthy execution time is allowed, using a simple link traversal-based querying approach does not provide adequate results for these queries.

Query	Local Graph Size	Result Size	Completion Time(minutes)
1	2064	141	
2	2250	112	
3	40767	26	
4	2944	28	
5	257	59	1.8
7	3	1	0.13
9	4056	0	
10	512	3	4.91
11	10908	520	

Table 1: FedBench Linked Data queries executed using the active discovery manager for 10 minutes. *Completion time refers to the time after which all URIs have been dereferenced, there are no more links to traverse, and therefore the query execution engine terminates early.*

4.3 Using SPARQL Endpoints only

In order to determine a baseline for the use of a single endpoint to answer the query, each query was sent to the set of endpoints used and the largest result sets and their response times listed in Table 2. It can be seen that Sindice is able to answer all the queries, providing in most cases a large number of results. Although the LIMIT keyword would reduce query response time if a known number of results are required, it is still difficult for a client to know exactly what the impact would be on processing time. Furthermore, although the DBpedia and SemanticWeb.org endpoints are expected to be in sync with the data they represent, an indexing approach such as Sindice may not be, as shown in [22].

4.4 Using a Hybrid Approach with Fixed Time Constraints

Queries were executed for 10 seconds using both the endpoint query manager and active discovery manager. All subqueries sent by the endpoint query manager were restricted to a result size of 100 using LIMIT in order to avoid time-consuming sub-queries. Table 3 summarises the number of results obtained, the size of the local graph and evaluation time (this refers to the use of the Jena query execution implementation to produce the result of the federated query from the local graph). Firstly it can be observed that regardless of the size of the local graph constructed within 10 seconds of query execution, the evaluation time was negligible in comparison. The largest evaluation time, 36 milliseconds, hardly makes a difference to a 10 second query response time. The approach returned results for all the queries execute, apart from the aforementioned queries 6 and 8.

Regarding the freshness of the results and the range of PLDs accessed, Table 4 shows that last modified dates of the RDF/XML pages retrieved were shown to be within the last 24 hours in 5 out of the 9 queries. Although this does not necessarily mean that the data has definitely changed within that time, it does indicate that there is the potential to retrieve fresher results than indexers which crawl the Semantic Web with less frequency, and may be of more benefit with queries executed over a broader set of Linked Data than the FedBench queries used here. The number of PLDs accessed was

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbprop: <http://dbpedia.org/property/> PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbowl: <http://dbpedia.org/ontology/>  PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>     PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

Evaluated queries:

```
1  SELECT * WHERE {
    ?paper <http://data.semanticweb.org/ns/swc/ontology#isPartOf>
           <http://data.semanticweb.org/conference/iswc/2008/poster_demo_proceedings> .
    ?paper <http://swrc.ontoware.org/ontology#author> ?p . ?p rdfs:label ?n
   }

2  SELECT * WHERE {
    ?proceedings <http://data.semanticweb.org/ns/swc/ontology#relatedToEvent>
                 <http://data.semanticweb.org/conference/eswc/2010> .
    ?paper <http://data.semanticweb.org/ns/swc/ontology#isPartOf> ?proceedings .
    ?paper <http://swrc.ontoware.org/ontology#author> ?p .
   }

3  SELECT * WHERE {
    ?paper <http://data.semanticweb.org/ns/swc/ontology#isPartOf>
           <http://data.semanticweb.org/conference/iswc/2008/poster_demo_proceedings> .
    ?paper <http://swrc.ontoware.org/ontology#author> ?p .
    ?p owl:sameAs ?x . ?p rdfs:label ?n .
   }

4  SELECT * WHERE {
    ?role <http://data.semanticweb.org/ns/swc/ontology#isRoleAt> <http://data.semanticweb.org/conference/eswc/2010> .
    ?role <http://data.semanticweb.org/ns/swc/ontology#heldBy> ?p .
    ?paper <http://swrc.ontoware.org/ontology#author> ?p .
    ?paper <http://data.semanticweb.org/ns/swc/ontology#isPartOf> ?proceedings .
    ?proceedings <http://data.semanticweb.org/ns/swc/ontology#relatedToEvent>
                 <http://data.semanticweb.org/conference/eswc/2010> .
   }

5  SELECT * WHERE {
    ?a dbowl:artist dbpedia:Michael_Jackson .
    ?a rdf:type dbowl:Album .
    ?a foaf:name ?n .
   }

7  SELECT * WHERE {
    <http://sws.geonames.org/2921044/> gn:parentFeature ?x.
    ?x gn:name ?n .
   }

9  SELECT * WHERE {
    ?x <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:FIFA_World_Cup-winning_countries> .
    ?p dbprop:managerclubs ?x . ?p foaf:name "Luiz Felipe Scolari"@en .
   }

10 SELECT * WHERE {
    ?n <http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Chancellors_of_Germany> .
    ?p2 owl:sameAs ?n . ?p2 <http://data.nytimes.com/elements/latest_use> ?u .
   }

11 SELECT * WHERE {
    ?x dbowl:team dbpedia:Eintracht_Frankfurt .
    ?x rdfs:label ?y . ?x dbowl:birthDate ?d . ?x dbowl:birthPlace ?p . ?p rdfs:label ?l .
   }
```

Unevaluated queries:

```
6  SELECT * WHERE {?director dbowl:nationality dbpedia:Italy .
    ?film dbowl:director ?director . ?x owl:sameAs ?film . ?x foaf:based_near ?y .
    ?y gn:officialName ?n .
   }

8  SELECT * WHERE {drugCategory:micronutrient . ?drug drugbank:casRegistryNumber ?id .
    ?drug owl:sameAs ?s . ?s foaf:name ?o . ?s skos:subject ?sub
   }
```

Figure 2: FedBench Linked Data Queries.

Query 7 was modified – the subject and object in the final triple pattern were swapped as no results were obtained originally. Where queries used the predicate `skos:subject`, this was changed to `dcterms:subject`. We believe these changes were necessary to reflect the evolving nature of the active data utilised by the FedBench Linked Data queries. The language extension `@en` was added to query 9 as without it no results were returned. Queries 6 and 8 were problematic for the following reasons. Query 6 attempts to join Italian film directors from DBpedia and the locations of their films joining triple pattens. Even when using a combination of SPARQL endpoints and link-traversal, we could not find any such films with `foaf:based_near` and `gn:officialName` properties in the Linked Movie Database or elsewhere. For Query 8 using DrugBank to list information about the micronutrient drug category, even when omitting the final triple pattern we were unable find any results. It is possible that the data over which they are designed to be executed has changed since the initial formulation of the queries.

Query	Local Graph Size	Active Discovery Triples	**Result Size**	Evaluation Time (ms)
1	711	17	**297**	36
2	210	17	**147**	9
3	649	274	**304**	18
4	415	296	**50**	7
5	420	4	**241**	13
7	4	0	**1**	3
9	263	252	**1**	3
10	123	65	**3**	4
11	684	189	**892**	36

Table 3: FedBench Linked Data queries executed using the proposed approach for 10 seconds. *The "Active discovery triples" column lists the number of triples that were added to the local graph by the active discovery manager. These may represent fresher results than those obtained from SPARQL endpoints.*

SemanticWeb.org		
Query	Response Time	Result Size
1	55614	6408
2	17328	1480
3	105298	13256
4	12956	1600
Sindice		
Query	Response Time	Result Size
1	84611	22428
2	12407	1587
3	330630	60849
4	27107	3000
5	34387	14736
7	10328	3870
9	1190	1
10	1082	14
11	512798	100000
DBpedia		
Query	Response Time	Result Size
5	885	74
9	592	1

Table 2: FedBench Linked Data queries executed using single SPARQL endpoints. *Where it was possible to obtain results for queries using a single endpoint, the number of results and response times in milliseconds are shown. Note that query 11 initially failed with a "509 Bandwidth Limit Exceeded" error when executed using Sindice, and was therefore re-run with a LIMIT 100000 clause in the query.*

small for most queries, and we anticipate that better results could be obtained by exploiting the parallelism of the active discovery manager for queries which access Linked Data over a more diverse set of domains.

5. CONCLUSIONS

An approach utilising hybrid query execution to support Linked Data query processing within fixed time bounds has been presented. Although the approach has a disadvantage in that complete query answering is not guaranteed, the advantages of the proposed approach are:

- **Fault tolerance:** the approach uses multiple data sources, if any one data source is unavailable the effects can be mitigated.
- **Freshness:** where SPARQL endpoints based on indexing systems may be out of date, RDF data from the Web of Linked Data may provide more up-to-date results.
- **Increased coverage:** the approach can potentially provide more results than any one source.
- **Mitigating usage restrictions:** where fair-use restrictions prevent complete result sets to be obtained from SPARQL endpoints, combining data from multiple sources can increase the number of results.

As the approach exploits parallelism to retrieve data from multiple endpoints and PLDs concurrently, we expect that better performance will be obtainable as the Linked Data becomes more widely adopted. The number of PLDs accessed by the active discovery manager for the FedBench queries was quite small, and we expect more impressive results for more diverse queries.

Future work will look at such queries and the use of emerging benchmarks for Linked Data query processing in conjunction with further optimisations, for example caching previously retrieved results to further enhance the effectiveness of the approach.

6. ACKNOWLEDGMENTS

This work is partly supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI grant numbers 24240015, 24680010 and 24700111.

Query	Freshness Same day	More than 1 month / Unknown	PLDs
1	**12**	14	xmlns.com, ontologydesignpatterns.org, olmedilla.info, semanticweb.org, eswc2006.org, ontoworld.org
2	**10**	2	semanticweb.org, ontoworld.org
3	**11**	28	xmlns.com, uni-leipzig.de, revyu.com olmedilla.info, semanticweb.org, olafhartig.de uni-koblenz.de, eswc2006.org, ontoworld.org
4	**7**	0	semanticweb.org
5	0	14	dbpedia.org
7	0	2	geonames.org
9	0	12	dbpedia.org
10	**3**	24	nytimes.com, zitgist.com, mpii.de dbpedia.org, freebase.com
11	0	12	dbpedia.org

Table 4: Freshness and PLDs of documents accessed by the active discovery manager. *The table summarises the gap between the last modified dates of RDF/XML pages accessed by the active discovery manager and the time at which they were accessed. 5 out of the 9 queries executed accessed an RDF/XML document that had a last modified date within the last 24 hours.*

7. REFERENCES

[1] Apache Jena. http://jena.apache.org/.

[2] DBPedia. http://dbpedia.org/.

[3] Describing Linked Datasets with the VoID Vocabulary (W3C Interest Group Note 03 March 2011). http://www.w3.org/TR/void/.

[4] Good Relations: The Web Vocabulary for E-Commerce. http://www.heppnetz.de/projects/goodrelations/.

[5] RDFa Primer. http://www.w3.org/TR/xhtml-rdfa-primer/.

[6] RDF/XML Syntax Specification. http://www.w3.org/TR/REC-rdf-syntax/.

[7] Sindice: The Semantic Web Index. http://sindice.com/.

[8] SPARQL 1.1 Federation Extensions. http://www.w3.org/2009/ sparql/docs/fed/gen.html.

[9] SQUIN - Query the Web of Linked Data. http://squin.sourceforge.net/.

[10] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ISWC'11, pages 18–34, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and Optimization of the SPARQL 1.1 Federation Extension. In G. Antoniou, M. Grobelnik, E. P. B. Simperl, B. Parsia, D. Plexousakis, P. D. Leenheer, and J. Z. Pan, editors, *ESWC (2)*, volume 6644 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011.

[12] K. G. Clark, L. Feigenbaum, and E. Torres. SPARQL Protocol for RDF. Technical report, W3C, 2008.

[13] S. H. Garlik, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/.

[14] O. Hartig and J.-C. Freytag. Foundations of traversal based query execution over linked data. In E. V. Munson and M. Strohmaier, editors, *HT*, pages 43–52. ACM, 2012.

[15] O. Hartig and A. Langegger. A Database Perspective on Consuming Linked Data on the Web. *Datenbank-Spektrum*, 10(2):57–66, 2010.

[16] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space.* Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[17] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax: http://www.w3.org/TR/rdf-concepts/. 2004.

[18] A. Langegger, A. Woss, and W. Bloch. A Semantic Web Middleware for Virtual Data Integration on the Web. In *5th European Semantic Web Conference (ESWC 2008)*, 2008.

[19] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[20] B. Quilitz and U. Leser. Querying Distributed RDF Data Sources with SPARQL. In *5th European Semantic Web Conference (ESWC 2008)*, 2008.

[21] M. Schmidt, O. GÃűrlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 585–600. Springer, 2011.

[22] J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira. Freshening up while Staying Fast: Towards Hybrid SPARQL Queries. In A. ten Teije, J. Völker, S. Handschuh, H. Stuckenschmidt, M. d'Aquin, A. Nikolov, N. Aussenac-Gilles, and N. Hernandez, editors, *EKAW*, volume 7603 of *Lecture Notes in Computer Science*, pages 164–174. Springer, 2012.