# Executing SPARQL Queries over the Web of Linked Data

Olaf Hartig[1], Christian Bizer[2], and Johann-Christoph Freytag[1]

[1] Humboldt-Universität zu Berlin
lastname@informatik.hu-berlin.de
[2] Freie Universität Berlin
firstname.lastname@fu-berlin.de

**Abstract.** The Web of Linked Data forms a single, globally distributed dataspace. Due to the openness of this dataspace, it is not possible to know in advance all data sources that might be relevant for query answering. This openness poses a new challenge that is not addressed by traditional research on federated query processing. In this paper we present an approach to execute SPARQL queries over the Web of Linked Data. The main idea of our approach is to discover data that might be relevant for answering a query during the query execution itself. This discovery is driven by following RDF links between data sources based on URIs in the query and in partial results. The URIs are resolved over the HTTP protocol into RDF data which is continuously added to the queried dataset. This paper describes concepts and algorithms to implement our approach using an iterator-based pipeline. We introduce a formalization of the pipelining approach and show that classical iterators may cause blocking due to the latency of HTTP requests. To avoid blocking, we propose an extension of the iterator paradigm. The evaluation of our approach shows its strengths as well as the still existing challenges.

## 1 Introduction

An increasing amount of data is published on the Web according to the Linked Data principles [1,2]. Basically, these principles require the identification of entities with URI references that can be resolved over the HTTP protocol into RDF data that describes the identified entity. These descriptions can include RDF links pointing at other data sources. RDF links take the form of RDF triples, where the subject of the triple is a URI reference in the namespace of one data source, while the object is a URI reference in the namespace of the other. The Web of Linked Data that is emerging by connecting data from different sources via RDF links can be understood as a single, globally distributed dataspace [3].

Querying this dataspace opens possibilities not conceivable before: Data from different data sources can be aggregated; fragmentary information from multiple sources can be integrated to achieve a more complete view. However, evaluating queries over the Web of Linked Data also poses new challenges that do not arise

A. Bernstein et al. (Eds.): ISWC 2009, LNCS 5823, pp. 293–309, 2009.

```
1  SELECT DISTINCT ?author ?phone WHERE {
2    <http://data.semanticweb.org/conference/eswc/2009/proceedings>
3                                                    swc:hasPart ?pub .
4    ?pub swc:hasTopic ?topic .
5    ?topic rdfs:label ?topicLabel .
6    FILTER regex( str(?topicLabel), "ontology engineering", "i" ) .
7
8    ?pub swrc:author ?author .
9    {?author owl:sameAs ?authAlt} UNION {?authAlt owl:sameAs ?author}
10
11   ?authAlt foaf:phone ?phone
12 }
```

**Fig. 1.** SPARQL query which asks for the phone numbers of people who authored an ontology engineering related paper at ESWC'09 (prefix declarations omitted)

in traditional federated query processing: Due to the openness of the dataspace, it is not possible to know all data sources that might be relevant for answering a query in advance.

Consider, for instance, the SPARQL query [4] in Figure 1 which asks for the phone number of people who authored an ontology engineering related paper at the European Semantic Web Conference 2009 (ESWC'09). This query cannot be answered from a single dataset but requires data from a large number of sources on the Web. For instance, the list of papers and their topics (cf. lines 2 to 4) is part of the Semantic Web Conference Corpus[1]; the names of the paper topics (cf. line 5) are provided by the sources authoritative for the URIs used to represent the topics; the phone numbers (cf. line 11) are provided by the authors. Hence, this kind of queries can only be answered using a method to execute queries without knowing the sources that contribute to the query result in advance. In this paper we present such a method.

The main idea of our approach is the asynchronous traversal of RDF links to discover data that might be relevant for a query during the query execution itself. Hence, the query engine executes each query over a growing set of potentially relevant data retrieved from the Web. Notice, in contrast to federated query processing [5] – which presumes each data source provides a query service – we do not distribute parts of the query evaluation. Instead, we only require the data sources to publish data following the Linked Data principles. This approach enables the execution of queries without knowing the sources that contribute to the query result in advance. Our main contributions are:

- an approach to execute SPARQL queries over the Web of Linked Data,
- a formal description of the realization of our approach with an iterator-based pipeline that enables an efficient query execution, and
- an extension of the iterator paradigm that avoids blocking of the query execution caused by waiting for data from the Web.

This paper is structured as follows. First, Section 2 gives an overview of our approach. In Section 3 we introduce an iterator-based evaluation of queries and

[1] http://data.semanticweb.org

present a formalism to describe this kind of evaluation. Section 4 discusses the application of an iterator-based query evaluation to our query execution approach and presents a strategy to execute these queries more efficiently. Even with these strategies, waiting for data from the Web may cause delays in query execution. Thus, in Section 5 we introduce an extension to the iterator paradigm that avoids blocking caused by these delays. An evaluation of our approach and the concepts to implement it is given in Section 6. Finally, we review related work in Section 7 and conclude in Section 8.

## 2 Overview of the Query Execution Approach

This section gives an informal overview of the proposed approach to execute SPARQL queries over the Web of Linked Data. SPARQL, the query language for RDF data [4], is based on graph patterns and subgraph matching. The basic building block from which more complex SPARQL query patterns are constructed is a basic graph pattern (BGP). A BGP is a set of triple patterns which are RDF triples that may contain query variables at the subject, predicate, and object position. During query evaluation solutions that bind values to the variables are determined.

To query the Web of Linked Data, we propose to intertwine query execution with the traversal of RDF links to discover data that might be relevant to answer the query. Using the data retrieved from looking up the URIs in a query as a starting point we evaluate parts of the query. The intermediate solutions resulting from this partial evaluation usually contain further URIs. These URIs link to additional data which may provide further solutions for the same or for other parts of the query. To determine results for the whole query we alternately evaluate query parts and dereference URIs. Hence, during query evaluation we continuously augment the queried dataset with potentially relevant data from the Web. The discovery of this data is driven by the URIs in intermediate results.

*Example 1.* The evaluation of the query in Figure 1 may start with RDF data retrieved from the Semantic Web Conference Corpus by dereferencing the URI identifying the ESWC'09 proceedings. This data contains a set of RDF triples that match the triple pattern in lines 2 and 3 of the query. The query engine generates a set of intermediate solutions from this set. Each of these solutions binds query variable `?pub` to the URI representing one of the papers in the ESWC'09 proceedings. Dereferencing these URIs yields RDF data about the papers including the topics of the publications. Hence, in this newly retrieved data the query engine finds matching triples for the pattern at line 4 with the given `?pub` binding. Based on these matches existing intermediate solutions can be augmented with bindings for variable `?topic`. Since the topics are also resources represented by URIs additional data will be added to the queried dataset. The query engine proceeds with the outlined strategy in order to determine solutions that are results of the whole query.

To consider all data that is available by traversing RDF links from matching triples our approach uses the following heuristic. Before we evaluate a triple

pattern in order to find matching triples in the local dataset we ensure that the local dataset contains at least all data retrievable from dereferencing all URIs that are part of the triple pattern. For instance, the evaluation of the triple pattern in line 4 of our sample query using the intermediate solutions from the triple pattern in lines 2 and 3 comprises the evaluation of multiple triple pattern, actually – one for each `?pub` binding. As discussed before, we dereference each URI bound to variable `?pub` before we evaluate the corresponding triple pattern. Notice, in addition to the data retrieved by dereferencing the URIs in a query as an initial seed it is also possible to load further RDF data in the local dataset before executing the query. This possibility allows to explicitly ensure considering data that must not be missed during query execution. Furthermore, reusing the same local dataset for different queries may yield more complete results since an already filled local dataset may contain data relevant to a query that would not be discoverable by executing the query itself. Such a shared dataset, however, requires the application of caching strategies which identify formerly retrieved data that might be stale and that has to be requested again.

Due to the openness and the widely distributed nature of the Web we cannot assume to find all data that is relevant to answer a query with our approach. Hence, we should never expect complete results. The degree of completeness depends on the structure of the network of RDF links as well as on the number of links. In a Web sparsely populated with links chances are slight to discover relevant data. Nonetheless, we are convinced the Web of Linked Data will rapidly grow in the coming years and so will the number and density of links. Further limitations of our approach are i) the need for initial URIs in the queries to start the link traversal, ii) infinite link discovery, iii) the retrieval of unforeseeable large RDF graphs from the Web, iv) URI dereferencing that takes unexpectedly long, v) Linked Data servers that put restrictions to clients such as serving only a limited number of requests per second, and vi) the possibility to overload a server. Some of these issues might be addressed with user-configurable options such as seed URIs, timeouts, and limits on traversal depth and filesizes. However, further investigation is required to find suitable defaults for these options and to address the listed issues in general.

## 3 Pipelining-Based Basic Graph Pattern Matching

We propose to implement our query execution approach using an iterator-based pipeline that enables an efficient, parallelized execution of queries. Introducing the pipelined evaluation of BGP queries over a dataset that is continuously augmented with potentially relevant data from the Web requires an understanding of the static case. Therefore, this section presents a formalism to describe a pipelining-based evaluation of BGPs over a fixed set of RDF graphs. While the SPARQL specification introduces different types of graph patterns we focus on BGPs, the fundamental building block, in this paper; the application of the presented concepts to more complex query patterns is subject to further research.

### 3.1 Solutions in SPARQL Query Execution

A formal description of our approach requires an explicit definition of the notion of a solution. Solutions in SPARQL are defined in the context of BGP matching where each solution basically represents a matching subgraph in the queried RDF graph. To describe these solutions the SPARQL specification [4] introduces a *solution mapping* that binds query variables to RDF terms such as URIs and literals. Solution mappings can be understood as a set of variable-term-pairs where no two pairs contain the same variable. The application of such a solution mapping $\mu$ to a BGP $b$, denoted with $\mu[b]$, implies replacing each variable in the BGP by the RDF term it is bound to in the mapping; unbound variables must not be replaced. Based on these mappings the specification defines solutions of BGPs for RDF graphs. Since our approach is based on a local dataset that contains multiple RDF graphs – one from each dereferenced URI – we slightly adjust this definition and introduce solutions of BGPs for sets of RDF graphs:

**Definition 1.** *Let $b$ be a BGP; let $\mathcal{G}$ be a set of RDF graphs. The solution mapping $\mu$ is a* ***solution*** *for $b$ in $\mathcal{G}$ if i) $\mu[b]$ is a subgraph of $\bigcup_{G \in \mathcal{G}} G$ and ii) $\mu$ does not map variables that are not in $b$.*

### 3.2 An Algorithm to Evaluate Basic Graph Patterns

During query processing a query is represented by a tree of logical operators. From this operator tree the query engine generates a query execution plan that implements the logical operators by physical operations. A well established approach to realize query plans is *pipelining* in which each solution produced by one operation is passed directly to the operation that uses it [6]. The main advantage of pipelining is the rather small amount of memory that is needed compared to approaches that completely materialize intermediate results. Pipelining in query engines is typically realized by a tree of *iterator*s that implement the physical operations [7]. An iterator is a group of three functions: `Open`, `GetNext`, and `Close`. `Open` initializes the data structures needed to perform the operation; `GetNext` returns the next result of the operation; and `Close` ends the iteration and releases allocated resources. An iterator allows a consumer to get the results of an operation separately, one at a time. In a tree of iterators the `GetNext` functions of an iterator typically call `GetNext` on the child(ren) of the iterator. Hence, a tree of iterators calculates solutions in a pull fashion.

Many in-memory RDF stores realize the evaluation of BGP queries by a tree of iterators as follows: Each iterator is responsible for a single triple pattern of the BGP $\{tp_1, \ldots, tp_n\}$. The iterators are chained together such that the iterator $I_i$ which is responsible for triple pattern $tp_i$ is the argument of the iterator $I_{i+1}$ responsible for $tp_{i+1}$. Each iterator returns solution mappings that are solutions for the set of triple patterns assigned to it and to its predecessors. For instance, the iterator $I_i$ returns solutions for $\{tp_1, \ldots, tp_i\}$ $(i \leq n)$. To determine these solutions the iterators basically execute the following three steps repeatedly: first, they consume the solution mappings from their direct predecessor; second, they apply these input mappings to their triple pattern; and, third, they try to

```
1  FUNCTION Open
2      LET G := the queried set of RDF graphs;
3      LET I_{i-1} := the iterator responsible for tp_{i-1};
4      CALL I_{i-1}.Open;
5
6      LET μ_cur := NotFound;
7      LET I_find := an iterator over an empty set of RDF triples;
8
9  FUNCTION GetNext
10     LET t := I_find.GetNext;
11     WHILE t = NotFound DO
12     {
13       LET μ_cur := I_{i-1}.GetNext;
14       IF μ_cur = NotFound THEN
15         RETURN NotFound;
16
17       LET I_find := an iterator over a set of all triples that match μ_cur[{tp_i}] in G;
18       LET t := I_find.GetNext;
19     }
20     LET μ' := the solution for μ_cur[{tp_i}] in G that corresponds to t;
21     RETURN μ_cur ∪ μ';
22
23 FUNCTION Close
24     CALL I_{i-1}.Close;
```

**Fig. 2.** Pseudo code notation of the `Open`, `GetNext`, and `Close` functions for an iterator $I_i$ that evaluates triple pattern $tp_i$

find triples in the queried RDF data that match the triple patterns resulting from the applications of the input mappings. Figure 2 illustrates the corresponding algorithm executed by the `GetNext` function of the iterators. Notice, the third step is realized with another type of iterator that returns the matching triples. We do not discuss this *helper iterator* in detail because its realization is not important for the concepts presented in this paper. Hence, in the remainder we simply assume the helper iterator iterates over a set of all RDF triples that match a triple pattern in a set of RDF graphs.

### 3.3 A Formalization of Iterator-Based Pipelining

The set of solutions provided by an iterator $I_i$ can be divided in subsets where each subset corresponds to one of the solutions consumed from the direct predecessor; i.e. each of these subsets contains all solutions that have been determined based on the same input solution. We denote these subsets with $Succ^{\mathcal{G}}(\mu, i)$.

$$Succ^{\mathcal{G}}(\mu, i) = \left\{\mu \cup \mu' \mid \mu' \text{ is a solution for } \mu[\{tp_i\}] \text{ in } \mathcal{G}\right\} \tag{1}$$

Notice, the $\mu'$s in Equation (1) correspond to the $\mu'$ in the algorithm (cf. line 20 in Figure 2). There is no $\mu'$ in Equation (1) that binds a variable which is already bound in $\mu$ because the application of $\mu$ to $\{tp_i\}$ yields $\{tp_i'\}$ where $tp_i'$ does not contain a variable bound in $\mu$. Hence, each $\mu'$ merely adds new bindings for variables not considered in $\mu$. For this reason it holds, if $\mu$ is a solution for $\{tp_1, \ldots, tp_{i-1}\}$ then each $\mu^* \in Succ^{\mathcal{G}}(\mu, i)$ is a solution for $\{tp_1, \ldots, tp_i\}$.

With $\Omega_i^{\mathcal{G}}$ we denote all solutions determined by the $i$th iterator. It holds

$$\Omega_i^{\mathcal{G}} = \begin{cases} Succ^{\mathcal{G}}(\mu_0, 1) & ; \text{ if } i = 1 \\ \bigcup_{\mu \in \Omega_{i-1}^{\mathcal{G}}} Succ^{\mathcal{G}}(\mu, i) & ; \text{ else} \end{cases} \tag{2}$$

where $\mu_0 = \varnothing$ is the empty solution mapping consumed by the first iterator.

**Proposition 1.** *$\Omega_n^{\mathcal{G}}$ is the result for BGP $\{tp_1, \dots, tp_n\}$ from RDF graph set $\mathcal{G}$.*

*Proof.* Each $\mu \in \Omega_1^{\mathcal{G}}$ is a solution from $\mathcal{G}$ for $\{tp_1\}$ because

$$\begin{aligned} \Omega_1^{\mathcal{G}} = Succ^{\mathcal{G}}(\mu_0, 1) &= \{\mu_0 \cup \mu' \mid \mu' \text{ is a solution for } \mu_0[\{tp_1\}] \text{ from } \mathcal{G}\} \\ &= \{\mu' \mid \mu' \text{ is a solution for } \{tp_1\} \text{ from } \mathcal{G}\} \end{aligned}$$

Let $i > 1$ and let $\Omega_{i-1}^{\mathcal{G}}$ all solutions from $\mathcal{G}$ for $\{tp_1, \dots, tp_{i-1}\}$. Due to Equation (1) it holds for each $\mu \in \Omega_{i-1}^{\mathcal{G}}$ that each $\mu_i \in Succ^{\mathcal{G}}(\mu, i)$ is a solution from $\mathcal{G}$ for $\{tp_1, \dots, tp_i\}$. Furthermore, $\Omega_i^{\mathcal{G}}$ is complete because it is the union of $Succ^{G}(\mu, i)$ for all possible $\mu \in \Omega_{i-1}^{\mathcal{G}}$. Hence, $\Omega_n^{\mathcal{G}}$ is the complete result from $\mathcal{G}$ for $\{tp_1, \dots, tp_n\}$. □

## 4 Evaluating Basic Graph Patterns over the Web

In this section we formalize our approach to evaluate BGP queries over the Web of Linked Data and we introduce strategies to execute these queries more efficiently.

To query the Web of Linked Data we cannot evaluate BGP queries over a fixed set of RDF graphs as introduced in the previous section. Instead, the queried dataset grows during the evaluation since we continuously add further, potentially relevant data by following the heuristic outlined in Section 2. The heuristic is based on the assumption that RDF graphs retrieved by looking up the URIs in a triple pattern might contain triples that match the triple pattern. Hence, we require that the local dataset contains these RDF graphs before we evaluate the triple pattern. More formally, we have to guarantee the following requirement.

**Requirement 1.** *The calculation of $Succ^{\mathcal{G}}(\mu, i)$ requires that*

1. *$deref\big(subj(\mu[tp_i])\big) \in \mathcal{G}$ if $subj(\mu[tp_i])$ is a URI,*
2. *$deref\big(pred(\mu[tp_i])\big) \in \mathcal{G}$ if $pred(\mu[tp_i])$ is a URI, and*
3. *$deref\big(obj(\mu[tp_i])\big) \in \mathcal{G}$ if $obj(\mu[tp_i])$ is a URI*

*where $\mu[tp]$ denotes the application of solution mapping $\mu$ to a triple pattern $tp$; $subj(tp)$, $pred(tp)$, and $obj(tp)$ denote the subject, predicate and object of a triple pattern $tp$, respectively; and $deref(u)$ represents the RDF graph that we retrieve by dereferencing the URI $u$.*

```
1  FUNCTION GetNext
2      LET t := I_find.GetNext;
3      WHILE t = NotFound DO
4      {
5        LET μ_cur := I_{i-1}.GetNext;
6        IF μ_cur = NotFound THEN
7          RETURN NotFound;
8
9        CALL EnsureRequirement for μ_cur[{tp_i}];
10       LET I_find := an iterator over a set of all triples that match μ_cur[{tp_i}] in G;
11       LET t := I_find.GetNext;
12     }
13
14     LET μ' := the solution for μ_cur[{tp_i}] in G that corresponds to t;
15     RETURN μ_cur ∪ μ';
```

**Fig. 3.** Pseudo code notation of the `GetNext` function for an iterator $I_i$ that evaluates triple pattern $tp_i$ over the Web of Linked Data

To guarantee Requirement 1 we adjust the `GetNext` function of our iterators as follows. Before the algorithm initializes the embedded helper iterator (cf. line 17 in Figure 2) it invokes a function called `EnsureRequirement`. This function checks the requirement and, if necessary, it dereferences URIs and waits until dereferencing has been finished. Figure 3 illustrates the adjusted `GetNext` function. Based on this adjustment the queried dataset $\mathcal{G}$ grows during the iterative calculation of $\Omega_n^{\mathcal{G}}$. The calculation of any solution subset $Succ^{\mathcal{G}}(\mu, i)$ uses the dataset $\mathcal{G}$ that is augmented with all RDF graphs retrieved for the calculation of previous subsets. However, each time the algorithm initializes $I_{\text{find}}$ (cf. line 10 in Figure 3) it uses an isolated snapshot of $\mathcal{G}$ that cannot be augmented by other iterators. This isolation avoids conflicts and an endless query execution because once the calculation of any subset $Succ^{\mathcal{G}}(\mu, i)$ has been initiated later additions to $\mathcal{G}$ are ignored for that calculation. The downside of this isolation is that the completeness of a query result depends on the order by which the iterators for the patterns in a query are chained. The development of concepts to find an order that is optimal with respect to maximizing result completeness is subject to further research.

The dereferencing requests in function `EnsureRequirement` of the adjusted iterators should be implemented by asynchronous function calls such that multiple dereferencing tasks can be processed in parallel. However, waiting for the completion of the dereferencing tasks in function `EnsureRequirement` delays the execution of the `GetNext` function and, thus, slows down query execution times. It is possible to address this problem with the following prefetching strategy.

Instead of dereferencing each URI at the time when the corresponding RDF graph is required we suggest to initiate the dereferencing task as soon as the URI becomes part of a solution. Considering that dereferencing requests are implemented by asynchronous function calls the query engine can immediately proceed the evaluation while the dereferencing tasks are executed separately. Whenever a subsequent iterator requires the corresponding dereferencing result chances are high that the dereferencing task has already been completed.

```
1  FUNCTION GetNext
2      LET t := I_find.GetNext;
3      WHILE t = NotFound DO
4      {
5        LET μ_cur := I_{i-1}.GetNext;
6        IF μ_cur = NotFound THEN
7          RETURN NotFound;
8
9        CALL EnsureRequirement for μ_cur[{tp_i}];
10       LET I_find := an iterator over a set of all triples that match μ_cur[{tp_i}] in G;
11       LET t := I_find.GetNext;
12     }
13
14     LET μ' := the solution for μ_cur[{tp_i}] in G that corresponds to t;
15
16     FOR EACH (var, val) ∈ μ' DO
17       IF val is a URI AND deref(val) ∉ G THEN Request the retrieval of deref(val);
18
19     RETURN μ_cur ∪ μ';
```

**Fig. 4.** Pseudo code notation of the `GetNext` function for an iterator $I_i$ that prefetches URIs during the evaluation of triple pattern $tp_i$

Figure 4 illustrates an adjusted `GetNext` function that realizes our URI prefetching strategy in lines 16 and 17. In Section 6.2 we analyze the impact of URI prefetching on query execution times.

## 5 Non-blocking Iterators

In this section we introduce an extension to the iterator paradigm. This extension prevents unnecessary long execution times caused by delays that cannot be avoided with URI prefetching.

URI prefetching as introduced in the previous section is an attempt to avoid delays during the execution of the `GetNext` function. However, this approach is not always sufficient as the following example demonstrates.

*Example 2.* Let $tp_i$ = (?x,?p,?o) the $i$th triple pattern in a BGP $\{tp_1, \ldots, tp_n\}$ where $1 < i \leq n$. Consider the iterator $I_i$ which is responsible for $tp_i$ is asked for the next solution by calling its `GetNext` function. Let the current helper iterator $I_{\text{find}}$ of $I_i$ be exhausted so that the algorithm enters the while loop (cf. line 3 in Figure 4) and requests the next input solution from the predecessor iterator $I_{i-1}$ (cf. line 5). Let $I_{i-1}$ return a solution mapping $\mu_{i-1}$ that binds variable ?x to URI *uri* where this binding has been determined by $I_{i-1}$; i.e. it holds (?x, *uri*) $\in \mu_{i-1}$ where $\mu_{i-1} \in Succ^{\mathcal{G}}(\mu, i-1) \subseteq \Omega^{\mathcal{G}}_{i-1}$ but (?x, *uri*) $\notin \mu$. Let $deref(uri) \notin \mathcal{G}$. In this case, $I_{i-1}$ has requested the retrieval of $deref(uri)$ just before it has returned $\mu_{i-1}$ to $I_i$. Since $I_i$ immediately calls `EnsureRequirement` for $\mu_{i-1}[\{tp_i\}]$ = (*uri*,?p,?o) it is very likely that the dereferencing of *uri* is still in progress. Hence, $I_i$ has to wait before it can initialize the next helper iterator and return the next solution.

As can be seen from Example 2 the prefetching of URIs does not prevent delays in the `GetNext` function in general. Unfortunately, an iterator that waits during the execution of `GetNext` causes a blocking of the whole query execution because the subsequent iterators wait for the result of `GetNext` and the predecessor iterators are not requested to do anything either. This problem might be addressed with program parallelism and the use of asynchronous pipelines [8]. According to this approach all iterators work in parallel; each pair of connected iterators shares a buffering queue to which the providing iterator asynchronously adds its intermediate solutions; the consuming iterator dequeues these solutions to process them. However, the realization of this approach would require a major rewrite of existing query engines that are based on synchronous pipelines. For this reason we propose an extension to the iterator paradigm which is compatible with the commonly used synchronous pipelining approach presented in this paper.

The core idea of our extension is to enable iterators to temporarily reject a solution consumed from its predecessor. Given such a possibility an iterator that finds Requirement 1 is not fulfilled for an input solution could reject that solution and ask the predecessor for another solution. To enable this possibility we add a new function, called `Reject`, to the iterator paradigm and we slightly extend the semantics of the `GetNext` function. The new function `Reject` treats the result that has most recently been provided by the `GetNext` function as if this result has never been provided by `GetNext`. This means `Reject` takes the rejected result back and keeps it for later requests. The extended `GetNext` function either returns the next result of the operation performed by the iterator or it returns one of the rejected results. Once a formerly rejected result is not being rejected again it must not be kept any further. Notice, we allow `GetNext` to decide nondeterministically to return a newly calculated result or to return a formerly rejected result. This approach provides more flexibility for realizations of our extended iterator paradigm. Analogously, we do not prescribe which of the rejected results have to be returned. However, in most cases it would probably be unwise to immediately reoffer a recently rejected result.

Figure 5 illustrates an application of the extended iterator paradigm to the iterators that evaluate BGP queries over the Web of Linked Data. Notice, these iterators call a function `CheckRequirement` which is similar to the function `EnsureRequirement` introduced in Section 4. Both functions check Requirement 1 and, if necessary, request the dereferencing of URIs. However, in contrast to `EnsureRequirement` the function `CheckRequirement` does not wait until the requested data has been retrieved, but, it returns an indication that either Requirement 1 is fulfilled or that the retrieval of data has been requested. In the latter case the algorithm in function `GetNext` rejects the current input solution from the predecessor iterator (cf. line 39 in Figure 5). Hence, the extended iterator paradigm allows to temporarily reject input solutions. This possibility to postpone the processing of certain solutions has a significant impact on query execution times as our evaluation in Section 6.2 illustrates.

```
1  FUNCTION Open
2        LET G := the queried set of RDF graphs;
3        LET I_{i-1} := the iterator responsible for tp_{i-1};
4        CALL I_{i-1}.Open;
5
6        LET μ_cur := NotFound;
7        LET I_find := an iterator over an empty set of RDF triples;
8
9        LET Ψ := an empty list; // used to keep rejected solutions
10       LET μ_last := NotFound; // used to hold the most recently provided solutions
11
12
13 FUNCTION GetNext
14       LET new := a randomly choosen element from {TRUE, FALSE};
15       IF Ψ is not empty AND new = FALSE THEN
16          LET μ_last := the first element in Ψ;
17          Remove μ_last from Ψ;
18          RETURN μ_last;
19
20       LET t := I_find.GetNext;
21       WHILE t = NotFound DO
22       {
23          LET μ_cur := I_{i-1}.GetNext;
24          IF μ_cur = NotFound THEN
25          {
26             IF Ψ is empty THEN
27                LET μ_last := NotFound;
28                RETURN μ_last;
29
30             LET μ_last := the first element in Ψ;
31             Remove μ_last from Ψ;
32             RETURN μ_last;
33          }
34
35          IF CheckRequirement for μ_cur[{tp_i}] returned TRUE THEN
36             LET I_find := an iterator over a set of all triples that match μ_cur[{tp_i}] in G;
37             LET t := I_find.GetNext;
38          ELSE
39             CALL I_{i-1}.Reject;
40       }
41
42       LET μ' := the solution for μ_cur[{tp_i}] in G that corresponds to t;
43
44       FOR EACH (var, val) ∈ μ' DO
45          IF val is a URI AND deref(val) ∉ G THEN Request the retrieval of deref(val);
46
47       LET μ_last := μ_cur ∪ μ';
48       RETURN μ_last;
49
50
51 FUNCTION Reject
52       IF μ_last ≠ NotFound THEN Append μ_last to Ψ;
53
54
55 FUNCTION Close
56       CALL I_{i-1}.Close;
```

**Fig. 5.** Pseudo code notation of the `Open`, `GetNext`, `Reject`, and `Close` functions for a non-blocking iterator $I_i$ that evaluates triple pattern $tp_i$

## 6 Evaluation

As a proof-of-concept, we implemented our approach to query the Web of Linked Data in the Semantic Web Client Library[2] (SWClLib). This library is available as Free Software; it is portable to any major platform due to its implementation in Java. Based on SWClLib we evaluate our approach in this section; we present real-world use cases and we evaluate the iterator-based implementation in a controlled environment.

### 6.1 Real-World Examples

To demonstrate the feasibility of our approach we tested it with the following four queries that require data from multiple data sources to be answered:

$Q1$: Return phone numbers of authors of ontology engineering papers at ESWC09
$Q2$: What are the interests of the people Tim Berners-Lee knows?
$Q3$: What natural alternatives can be used instead of the drug "Varenicline"?
$Q4$: Return the cover images of soundtracks for movies directed by Peter Jackson.

Our demo page[3] provides the SPARQL representations of the test queries and it allows to execute these queries using SWClLib. In Table 1 we present average measures for these executions; the table illustrates the number of query results for each query, the number of RDF graphs retrieved during query execution, the number of servers from which the graphs have been retrieved, and the execution time. The first query, $Q1$, which is the sample query introduced in Section 1 (cf. Figure 1) has been answered using the Semantic Web Conference Corpus, DBpedia, and personal FOAF profiles. Answering query $Q2$ required data from Tim Berners-Lee as well as data from all the people he knows. Query $Q3$ was answered with data retrieved from the DrugBank database, the Diseasome dataset, and the Traditional Chinese Medicine dataset. The results for $Q4$ have been constructed with data from LinkedMDB and the MusicBrainz dataset.

In addition to the test queries we developed Researchers Map [9] to demonstrate that our approach is suitable for applications that consume Linked Data. Researchers Map is a simple mash-up that is solely based on the results of queries evaluated over the Web of Linked Data. The main feature of Researchers Map

**Table 1.** Statistics about the test queries

| **query** | $Q1$ | $Q2$ | $Q3$ | $Q4$ |
|---|---|---|---|---|
| **# of results** | 2 | 27 | 7 | 5 |
| **# of retrieved graphs** | 297 | 85 | 28 | 442 |
| **# of accessed servers** | 16 | 46 | 6 | 6 |
| **execution time** | 3min 47sec | 1min 11sec | 0min 46sec | 1min 24sec |

[2] http://www4.wiwiss.fu-berlin.de/bizer/ng4j/semwebclient/
[3] http://squin.informatik.hu-berlin.de/SQUIN/

is a map of professors from the German database community; the list of professors in the map can be filtered by research interests; selecting a professor opens a list of her/his publications. This application relies on data published by the professors as well as on data from DBpedia and the DBLP dataset.

### 6.2 The Impact of URI Prefetching and Non-blocking Iterators

Based on the SWClLib we conducted experiments to evaluate the presented concepts for an iterator-based realization of our query execution approach. For our tests we adopt the Berlin SPARQL Benchmark (BSBM) [10]. The BSBM executes a mix of 12 SPARQL queries over generated sets of RDF data; the datasets are scalable to different sizes based on a scaling factor. Using these datasets we set up a Linked Data server[4] which publishes the generated data following the Linked Data principles. With this server we simulate the Web of Linked Data in our experiments.

To measure the impact of URI prefetching and of our iterator paradigm extension we use the SWClLib to execute the BSBM query mix over the simulated Web. For this evaluation we adjust the SPARQL queries provided by BSBM in order to access our simulation server. We conduct our experiments on an Intel Core 2 Duo T7200 processor with 2 GHz, 4 MB L2 cache, and 2 GB main memory. Our test system runs a recent 32 bit version of Gentoo Linux with Sun Java 1.6.0. We execute the query mix for datasets generated with scaling factors of 10 to 60; these datasets have sizes of 4,971 to 26,108 triples, respectively. For each dataset we run the query mix 6 times where the first run is for warm up and is not considered for the measures.

Figure 6(a) depicts the average times to execute the query mix with three different implementations of SWClLib: i) without URI prefetching, ii) with prefetching, and iii) with the extended iterators that postpone the processing of input solutions. As can be seen from the measures URI prefetching reduces the query execution times to about 80%; our non-blocking iterators even halve the time.

The chart in Figure 6(b) puts the measures in relation to the time it takes to execute the query mixes without the need to retrieve data from the Web. We obtained this optimum by executing each query twice over a shared dataset; we measured the second executions which did not require to look up URIs because all data has already been retrieved in the first pass. These measures represent only the time to actually evaluate the queries as presented in Section 3. Hence, these times are a lower bound for possible optimizations to the iterator-based execution of our approach to query the Web of Linked Data. Using this lower bound we calculate the times required for data retrieval in the three implementations. These times are the differences between execution times measured for the three implementations and the lower bound, respectively. Figure 6(b) depicts these numbers which illustrate the significant impact of the possibility to

[4] Our server is based on RAP Pubby which is available from http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/tutorial/RAP_Pubby.htm

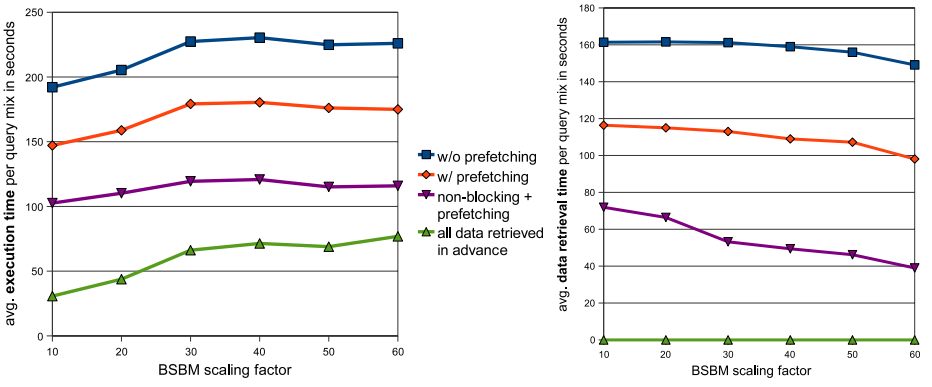


**Fig. 6.** Average times to execute the BSBM query mix with the SWClLib over a simulated Web of different sizes measured without URI prefetching, with prefetching, and with the extended iterators that temporarily reject input solutions, respectively

postpone the processing of certain input solutions in our non-blocking iterators. The chart additionally illustrates that the data retrieval times compared to the whole query execution time decreases for larger datasets, in particular in the case of the non-blocking iterators.

## 7 Related work

In this paper we present an approach to execute queries over the Web of Linked Data. Different solutions with a similar goal have been proposed before. These approaches can be classified in two categories: query federation and data centralization.

Research on federated query processing has a long history in database research. Sheth and Larson [5] provide an overview of the concepts developed in this context. Current approaches adapt these concepts to provide integrated access to distributed RDF data sources on the Web. The DARQ engine [11], for instance, decomposes a SPARQL query in subqueries, forwards these subqueries to multiple, distributed query services, and, finally, integrates the results of the subqueries. Very similar to the DARQ approach the SemWIQ [12] system contains a mediator service that transparently distributes the execution of SPARQL queries. Both systems, however, do not actively discover relevant data sources that provide query services.

In contrast to federation based systems the idea of centralized approaches is to provide a query service over a collection of Linked Data copied from different sources on the Web. For instance, OpenLink Software republishes the majority of the datasets from the Linking Open Data community project[5] on a

[5] http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

single site[6]; users can issue SPARQL queries over the whole set of mirrored datasets. As with the federation based systems this approach executes queries only on selected datasets that are part of the collection. Another form of the centralized approach are search engines for the Web of Linked Data such as Sindice [13], Swoogle [14], and Watson [15]. These search engines crawl the Web by following RDF links, index discovered data, and provide query interfaces to their indexes. In contrast to our approach where the queried data is discovered and retrieved at query execution time the crawlers collect the queried data in advance. Due to the possibility to consider data that would not be discovered by our method a crawling-based approach might return more complete query results. However, setting up an index that covers large amounts of data on the Web requires much more resources (e.g. storage, compute power, administrative personnel) than is needed to apply our approach. Furthermore, our method is superior with respect to the timeliness of query results because we only use that data that is available at the time of query execution.

The idea of looking up URIs during application runtime as in our approach has first been proposed by Berners-Lee et al. [16]. The authors outline an algorithm that traverses RDF links in order to obtain more data about the resources presented in the Tabulator Linked Data browser. Our approach is based on the idea of Berners-Lee et al. We integrate this idea in the process of query execution and, thus, resolve the need to implement link traversal algorithms in each application that consumes Linked Data.

## 8 Conclusion

In this paper we introduce an approach to query the Web of Linked Data and we present concepts and algorithms to implement this approach. Our approach is based on traversing RDF links to discover data that might be relevant for a query during the query execution itself. We propose to implement this idea with an iterator-based pipeline and a URI prefetching approach to execute queries efficiently. To improve the performance of query execution even more the paper introduces an extension to the iterator paradigm that allows to temporarily reject certain input results. We provide the Semantic Web Client Library as a first prototype that implements our approach; an application that uses this library demonstrates the feasibility of our idea.

Our approach benefits from a high number of links in the Web of Linked Data; the more links exist the more complete results can be expected because more relevant data might be discovered. In addition to relying on a dense network of links, sharing the queried dataset and in advance crawling show great promise to improve the completeness of the results. In both approaches query execution does not have to start on an empty dataset. Instead, potentially relevant data might already be available. Hence, we are investigating possibilities to combine these techniques with our approach. Such a combination introduces the need

[6] http://lod.openlinksw.com

for a proper caching solution with suitable replacement strategies and refreshing policies. We are currently working on these issues.

The openness of the Web of Linked Data holds an enormous potential which could enable users to benefit from a virtually unbound set of data sources. However, this potential will become available not until applications take advantage of the characteristics of the Web which requires approaches to discover new data sources by traversing data links. This paper presents a proof of concept to enable this evolutionary step.

## References

1. Berners-Lee, T.: Design Issues: Linked Data. (retrieved May 25, 2009), http://www.w3.org/DesignIssues/LinkedData.html
2. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Journal on Semantic Web and Information Systems (in press, 2009)
3. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: A new abstraction for information management. SIGMOD Record 34(4), 27–33 (2005)
4. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (January 2008), http://www.w3.org/TR/rdf-sparql-query/ (retrieved June 11, 2009)
5. Sheth, A.P., Larson, J.A.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Computing Surveys 22(3), 183–236 (1990)
6. Garcia-Molina, H., Widom, J., Ullman, J.D.: Database Systems: The Complete Book. Prentice-Hall, Inc., Upper Saddle River (2002)
7. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys 25(2), 73–169 (1993)
8. Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P.: Parallelism in relational data base systems: Architectural issues and design approaches. In: Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems (DPDS), pp. 4–29. ACM, New York (1990)
9. Hartig, O., Mühleisen, H., Freytag, J.-C.: Linked data for building a map of researchers. In: Proceedings of 5th Workshop on Scripting and Development for the Semantic Web (SFSW) at ESWC (June 2009)
10. Bizer, C., Schultz, A.: Benchmarking the performance of storage systems that expose SPARQL endpoints. In: Proceedings of the Workshop on Scalable Semantic Web Knowledge Base Systems at ISWC (October 2008)
11. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 524–538. Springer, Heidelberg (2008)
12. Langegger, A., Wöß, W., Blöchl, M.: A semantic web middleware for virtual data integration on the web. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 493–507. Springer, Heidelberg (2008)
13. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: A document-oriented lookup index for open linked data. International Journal of Metadata, Semantics and Ontologies 3(1) (2008)

14. Ding, L., Finin, T.W., Joshi, A., Pan, R., Cost, R.S., Peng, Y., Reddivari, P., Doshi, V., Sachs, J.: Swoogle: A search and metadata engine for the semantic web. In: Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM), November 2004, pp. 652–659. ACM, New York (2004)
15. d'Aquin, M., Motta, E., Sabou, M., Angeletou, S., Gridinoc, L., Lopez, V., Guidi, D.: Toward a new generation of semantic web applications. IEEE Intelligent Systems 23(3), 20–28 (2008)
16. Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., Sheets, D.: Tabulator: Exploring and analyzing linked data on the semantic web. In: Proceedings of the 3rd Semantic Web User Interaction Workshop (SWUI) at ISWC (November 2006)