Chapter 8

Linked Data Query Processing based on Link Traversal

Olaf Hartig
University of Waterloo
David R. Cheriton School of Computer Science
Waterloo, ON, Canada

8.1 Introduction

The execution of SPARQL queries over Linked Data readily available from a large number of sources provides enormous potential. Consider, for instance, the following SPARQL query 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 (e.g., in

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

their FOAF profiles). Hence, this kind of queries can only be answered using an approach for executing queries over Linked Data from multiple sources.

```
SELECT DISTINCT ?author ?phone WHERE {
  <http://data.semanticweb.org/conference/eswc/2009/proceedings>
                                                swc:hasPart ?pub .
  ?pub swc:hasTopic ?topic .
  ?topic rdfs:label ?topicLabel .
  FILTER regex( str(?topicLabel), "ontology engineering", "i" ) .

  ?pub swrc:author ?author .
  {?author owl:sameAs ?authAlt} UNION {?authAlt owl:sameAs ?author}

  ?authAlt foaf:phone ?phone }
```

An approach that enables the execution of such queries is to populate a centralized repository similar to the collection of Web documents managed by search engines for the Web. The database management systems for RDF data discussed in previous chapters of this book provide a basis for this approach.By using such a centralized repository it is possible to provide almost instant query results. This capability comes at the cost of setting up and maintaining the repository. Furthermore, users of such an interface for querying Linked Data are restricted to the portion of the Web of Data that has been copied into the repository. For instance, if we aim to answer our example query using a repository that lacks, e.g., some authors' FOAF profiles (or the most recent version thereof), we may get an answer that is incomplete (or outdated) w.r.t. all Linked Data available on the Web.

In this chapter we adopt an alternative view on querying Linked Data [35]. We conceive the Web of Data as a distributed database system. Querying the Web of Data itself opens possibilities not conceivable before; it enables users to benefit from a virtually unbounded set of up-to-date data.

However, the Web of Data is different from traditional distributed database systems: Usually, the latter assume data-local query processing functionality. Although the SPARQL protocol presents a commonly accepted standard for exposing such a functionality, we cannot generally assume that all publishers provide a SPARQL endpoint for their datasets. In contrast, while the Linked Data principles present a simple publishing method that can be easily added to existing workflows for generating HTML pages[2], providing and maintaining a (reliable) SPARQL endpoint presents a significant additional effort that not all publishers are willing (or able) to make. For instance, not many people expose their FOAF profile via a SPARQL endpoint (which renders a query execution approach that relies on such endpoints unsuitable for our example query). Therefore, in this chapter we understand a *Linked Data query* as a query that ranges over data that can be accessed by dereferencing URIs (a

[2]Using the RDFa standard, Linked Data can even be embedded in HTML documents [1], allowing publishers to serve a single type of document for human and machine consumption.

formal definition follows shortly). Consequently, *Linked Data query execution* relies solely on the Linked Data principles as introduced in Section 1.6.2.

Further distinguishing characteristics of the Web of Data are its unbounded nature and the lack of a complete database catalog. Due to these characteristics it is impossible to know all data sources that might contribute to the answer of a query. In this context, traditional query execution paradigms are insufficient because those assume that the query execution system (or the user) has information about the existence of any potentially relevant data source.

In this chapter we introduce a novel query execution paradigm that is tailored to the Web of Data. The general idea of this approach, which we call *link traversal based query execution* (or *LTBQE* for short), is to intertwine the construction of query results with the traversal of data links in order to discover data that might be relevant to answer the query.

Example 12. *A link traversal based query execution of our example query may start with some data retrieved from the Semantic Web Conference Corpus by dereferencing the URI that identifies 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 solution mappings from this set. Each of these solution mappings binds query variable* `?pub` *to the URI representing one of the papers in the ESWC'09 proceedings. Dereferencing these URIs yields Linked 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 previously generated solution mappings can be augmented with bindings for variable* `?topic`*. Since the topics are also denoted by URIs, additional data can be retrieved to generate bindings for* `?topicLabel`*. The query engine proceeds with the outlined strategy to eventually determine solution mappings that cover the whole query pattern and, thus, can be reported as solutions of the query result.*

The remainder of this chapter is organized as follows: As a theoretical foundation we first introduce a well-defined query semantics that allows us to use SPARQL as a query language for Linked Data queries. Thereafter, we introduce the core principles of the LTBQE paradigm and discuss merits and limitations of these principles. Finally, we focus on a particular example of an implementation of the LTBQE paradigm, for which we provide a more detailed discussion. This approach applies the well-known iterator model to implement the LTBQE paradigm.

8.2 Theoretical Foundations

To define Linked Data query execution approaches, discuss their respective merits and limitations, and compare them, in a meaningful way, we need a precise definition of what the supported queries are and what the expected results for such queries are. Hence, we need a well-defined query semantics.

In this chapter we focus on queries expressed using SPARQL. However, the original semantics of this query language assumes queries over a-priory defined sets of RDF triples (recall the definitions in Section 1.5.3 of the first chapter in this book). Hence, to use SPARQL as a language for queries over Linked Data *on the Web* we have to adjust the semantics by redefining the scope for evaluating SPARQL expressions. As a basis for such an adjustment we need a data model that formally captures the concept of a Web of Data. In this section we introduce these theoretical foundations. For simplicity, we assume a static view of the Web; that is, no changes are made to the data on the Web during the execution of a query.

8.2.1 Data Model

We represent a Web of Data[3] as a set of abstract symbols that is accompanied by two mappings. We call these symbols *LD document*s and use them to formally capture the concept of Web documents that can be obtained by dereferencing URIs according to the Linked Data principles. Consequently, the two accompanying mappings model the fact that we may obtain an RDF graph from such a document and that we may retrieve such documents by dereferencing URIs. Then, the definition of a Web of Data is given as follows:

Definition 15. *Let* $\mathbf{U}$ *be the set of all URIs (per Definition 1 in Chapter 1) and let* $\mathbf{T}$ *be the set of all RDF triples (per Definition 1 in Chapter 1). A* ***Web of Data*** *is a tuple* $W = (D, data, adoc)$ *where* D *is a set of LD documents; data is a total mapping:* $D \rightarrow 2^{\mathbf{T}}$ *such that* $data(d)$ *is finite for all* $d \in D$*; and adoc is a partial, surjective mapping:* $\mathbf{U} \rightarrow D$*.*

Given a Web of Data $W = (D, data, adoc)$ we say that an LD document $d \in D$ *describes* the resource identified by a URI $u \in \mathbf{U}$ if there exists an RDF triple $t = (s, p, o)$ such that i) $s = u$ or $o = u$ and ii) this triple t is contained in the RDF graph that can be obtained from LD document d; i.e., $t \in data(d)$. Clearly, for any URI $u \in \mathbf{U}$ there might be multiple LD documents in D that describe the resource identified by u. Moreover, in practice some HTTP URIs can be dereferenced to retrieve what may be understood as *authoritative*

[3]In this chapter we overload the term "*Web of Data*" to homonymously refer to the mathematical structure that we define as well as to the World Wide Web which presents an actual implementation of such a structure. Nonetheless, it should be clear from the context which meaning each use of the term refers to.

documents for these URIs. Mapping *adoc* captures this relationship. Note that this mapping is intentionally not required to be injective because dereferencing different URIs may result in the retrieval of the same document. On the other hand, not all URIs can be dereferenced; hence, mapping *adoc* is not total.

We can now define the notion of a Linked Data query formally:

Definition 16. *A* ***Linked Data query*** *$\mathcal{Q}$ is a total function over the set of all possible Webs of Data (i.e., all 3-tuples that satisfy Definition 15).*

Note that this definition of Linked Data queries is intentionally general; this generality allows for queries expressed in different query languages, different query semantics, and different types of query results.[4] In this chapter, we focus on *SPARQL based Linked Data queries*, that are, Linked Data queries (per Definition 16) that have the following two properties:

1. These queries are expressed using a SPARQL query pattern (per Definition 12 in Chapter 1); and
2. any query result for such a query must be a set of SPARQL solution mappings (per Definition 8 in Chapter 1); more precisely, the codomain of these queries is $2^{\mathbf{M}}$ where $\mathbf{M}$ denotes the infinite set of all possible solution mappings.

8.2.2 Full-Web Semantics for SPARQL

We now introduce query semantics for SPARQL based Linked Data queries. To this end, we adapt the usual SPARQL semantics from Chapter 1 such that SPARQL query patterns can be used as Linked Data queries. The most straightforward approach for such an adaptation is to assume a SPARQL dataset whose default graph consists of *all* RDF triples that exist in the queried Web of Data. Hereafter, we refer to this adaptation as *full-Web semantics* and call the resulting SPARQL based Linked Data queries *SPARQL$_{\mathsf{LD}}$ queries*.

To define these queries formally we denote the set of all RDF triples in a Web of Data $W = (D, data, adoc)$ by AllData(W); thus, it holds that:[5]

$$\mathrm{AllData}(W) = \biguplus_{d \in D} data(d)\,.$$

Then, defining SPARQL$_{\mathsf{LD}}$ queries is trivial and makes use of SPARQL query patterns, their execution function (per Definition 14; cf. Chapter 1), and the concept of a SPARQL dataset[6] (see Definition 6):

[4]An alternative to expressing Linked Data queries using SPARQL is the Linked Data query language NautiLOD as introduced in the following chapter. Query results in the case of NautiLOD are sets of URIs.

[5]Recall that the operator $\uplus$ denotes an RDF merge (see Definition 4 in Chapter 1).

[6]In this chapter we denote SPARQL datasets by DS because the symbol D is already reserved for the set of LD documents in a Web of Data.

Definition 17. *The* ***SPARQL***$_{\textsf{LD}}$ ***query*** *that uses SPARQL query pattern* P*, denoted by* $\mathcal{Q}^P$*, is a Linked Data query that, for any Web of Data* W*, is defined by* $\mathcal{Q}^P(W) := [\![P]\!]_{DS}$ *where* $DS = \{\text{AllData}(W)\}$*.*

To define a query semantics for Linked Data queries we may assume complete knowledge of the three elements that capture a queried Web formally (i.e., D, *data*, and *adoc*). An actual system that accesses an implementation of a Web of Data such as the WWW cannot have such knowledge. In contrast, to such a system the Web appears as an unbounded space: Without complete access to mapping *adoc* –in particular, dom(*adoc*)– potentially any HTTP URI may allow the system to retrieve Linked Data. However, the set of such URIs is infinite and, thus, the system would have to spend an infinite number of computation steps dereferencing all these URIs in order to guarantee that it has seen all of dom(*adoc*) and, thus, disclosed mapping *adoc* completely. Even if, at some point during this process, the system would –by chance– have dereferenced all URIs $u \in \text{dom}(adoc)$, it cannot know that this is the case. Therefore, in practice we cannot assume to ever have a complete list of all URIs based on which we would retrieve all Linked Data (even under our assumption that the Web of Data is static). Consequently, we also cannot assume that any system ever has access to all Linked Data that is or was (openly) available on the WWW at a certain point in time.

As a consequence of these limited data access capabilities *not any* approach for executing SPARQL$_{\textsf{LD}}$ queries over a Web of Data such as the WWW can guarantee complete query results. While a formal verification of this limitation is out of the scope of this chapter, we refer to Hartig's analysis [34]. The author introduces an abstract computation model that formally captures the data access capabilities of systems that aim to compute functions over a Web of Data such as the WWW. Based on this model the author formally analyzes feasibility and limitations of computing SPARQL based Linked Data queries. For full-Web semantics this analysis shows that there does not exist a single satisfiable SPARQL$_{\textsf{LD}}$ query for which there exists a sound and complete computation that terminates after a finite number of computation steps.

We emphasize, however, that instead of the presented full-Web semantics a SPARQL based Linked Data query may also be interpreted under an alternative, more restrictive query semantics. Several such semantics have been proposed [13, 31, 34], each of which restricts the range of Linked Data queries to a well-defined part of the queried Web of Data. Examples are query semantics that use a certain notion of navigational reachability to specify what part of a queried Web of Data needs to be considered for a given query. For a particular family of such reachability based semantics computational properties have been studied in the aforementioned analysis [34].

Due to space constraints we do not discuss any of these alternative query semantics here. Instead, for the purpose of introducing link traversal based query execution in the following sections, we assume full-Web semantics (with the caveat that completeness of query results cannot be guaranteed).

8.3 Principles and Characteristics of LTBQE

Link traversal based query execution (LTBQE) is a novel query execution paradigm tailored for Linked Data queries. In this section we discuss the following three core principles that are characteristic of any LTBQE approach:

Live exploration based data retrieval: For the execution of Linked Data queries it is necessary to retrieve data by dereferencing URIs. Consequently, any approach for executing Linked Data queries needs to prescribe a strategy for selecting those URIs that the query execution system dereferences during the execution of a given query. The strategy adopted by LTBQE is to explore the queried Web of Data by traversing data links *at query execution time*. More precisely, as an integral part of executing a Linked Data query, an LTBQE system performs a recursive URI lookup process. The starting point for this process is a set of *seed URIs*; these seed URIs may be the URIs mentioned in the given query, or they are specified as an accompanying parameter for the query. While the data retrieved during such a recursive live exploration process allows for a discovery of more URIs to look up, it also provides the basis for constructing the query result.[7] (A different strategy to retrieve data for executing Linked Data queries is to dereference a set of URIs selected from a pre-populated index. A discussion of this strategy can be found in Chapter **??** of this book.)

A live exploration based system may not need to dereference all URIs discovered. Instead, certain live exploration approaches may (directly or indirectly) introduce criteria to decide which of the discovered URIs are scheduled for lookup. For instance, approaches designed to support one of the more restrictive query semantics (mentioned in Section 8.2.2), may ignore any URI whose lookup exceeds the part of the Web that is relevant according to the semantics.

Integration of data retrieval and result construction: The actual process of executing a Linked Data query may consist of two separate phases: During the first phase a query execution system selects URIs and uses them to retrieve data from the queried Web; during a subsequent, second phase the system generates the query result using the data retrieved in the first phase. Instead of separating these two phases, a fundamental principle of LTBQE is to integrate the retrieval of data into the result construction process.

Combining live exploration and integrated execution: We emphasize that the design decision to integrate data retrieval and result construction is orthogonal to deciding what data retrieval strategy to use for

[7]To avoid blank-node label conflicts the query engine must ensure that distinct blank-node labels are used for each RDF graph discovered during the live exploration process.

such an integrated execution. Consequently, the combination of live exploration based data retrieval with the idea of an integrated execution is what ultimately characterizes the LTBQE paradigm.

In the following we first discuss separately the merits and the limitations of live exploration and of an integrated execution. Afterwards, we elaborate on how these ideas can be combined into an LTBQE strategy.

8.3.1 Live Exploration

The most important characteristic of live exploration based data retrieval is the possibility to use data from initially unknown data sources. This characteristic allows for serendipitous discovery and, thus, enables applications that tap the full potential of Linked Data on the WWW.

Another characteristic is that a live exploration based query execution system does not require any a-priori information about the queried Web of Data. Consequently, such a system might readily be used without having to wait for the completion of an initial data load phase or any other type of preprocessing. Hence, this characteristic makes live exploration based approaches (and, thus, LTBQE) most suitable for an "on-demand" querying scenario.

On the downside, it is inherent in the recursive URI lookup process that access times for data add up. Possibilities for parallelizing data retrieval are limited because relevant URIs only become available incrementally. Moreover, the recursive URI lookup process may not even terminate at all if we assume the queried Web of Data is infinitely large due to data generating servers [34].

Another limitation of live exploration based data retrieval is its dependency on the structure of the network of data links as well as on the number of links. In a Web sparsely populated with links chances are low to discover relevant data. While such a limitation is not an issue for queries under the aforementioned reachability based query semantics (cf. Section 8.2.2), systems that aim to support full-Web semantics may report more complete results for certain queries if they use another data retrieval strategy. For instance, a system that uses the index based strategy discussed in Chapter **??** may be able to compute some solutions of a query result that a live exploration based system cannot compute; this is the case if some data necessary for computing these solutions cannot be discovered by link traversal. On the other hand, a live exploration based system may discover URIs that are not mentioned in the index based system's index; the data retrieved by dereferencing these URIs may allow the live exploration based system to compute certain query solutions that the index based system cannot compute. Hence, a general statement about the superiority of the live exploration strategy over the index based strategy (or vice versa) w.r.t. result completeness is not possible in the context of full-Web semantics.

In its purest form, the live exploration strategy assumes query execution systems that do not have any a-priori information about the Web of Data and begin each query execution with an empty query-local dataset. It is also possible, however, that a query execution system uses the query-local dataset that it populated during the execution of a query as a basis for executing subsequent queries. Such a reuse can be beneficial for two reasons [32]: 1) it can improve query performance because it reduces the need to retrieve data multiple times; 2) assuming full-Web semantics, it can provide for more complete query results, calculated based on data from data sources that would not be discovered by a live exploration with an initially empty query-local dataset. However, since reusing the query-local dataset for the execution of multiple queries is a form of data caching, it requires suitable caching strategies. In particular, any system that keeps previously retrieved data has to apply an appropriate invalidation strategy; otherwise it could lose the advantage of up-to-date query results. As an alternative to caching retrieved data it is also possible to only keep a summary of the data or certain statistics about it. Such information may then be used to guide the execution of later queries (as in the case of index based source selection discussed in Chapter **??**).

8.3.2 Integrated Execution

The idea to intertwine the retrieval of data and the construction of query results may allow a query execution system to report first solutions for a (monotonic) query early, that is, before data retrieval has been completed.

Furthermore, this integrated execution strategy may be implemented in a way that requires significantly less query-local memory than any approach that separates data retrieval and result construction into two, consecutive phases; this may hold in particular for implementations that process retrieved data in a streaming manner and, thus, do not require to store all retrieved data until the end of a query execution process. For instance, Ladwig and Tran introduce such an implementation approach for LTBQE [49, 50].

8.3.3 Combining Live Exploration and Integrated Execution

While the two, previously discussed strategies, live exploration and integrated execution, should be understood as independent principles for designing a Linked Data query execution approach, they almost naturally fit together. Combining these two principles is what characterizes the LTBQE paradigm. Different approaches for such a combination have recently been studied in the literature; a manifold of other approaches is possible. In the remainder of this section we provide an overview on such approaches. In particular, we first outline a naive example of an LTBQE approach in order to illustrate the idea of combining live exploration based data retrieval with the integrated execution strategy. Thereafter, we refer to particular LTBQE approaches proposed in the literature.

Using a set of seed URIs as a starting point, a query execution system may alternate between two types of execution stages, that are, link traversal stages and result computation stages. Each link traversal stage consists of dereferencing URIs that the system finds in the data retrieved during the previous link traversal stage. During the result computation stage that follows such a link traversal stage the system generates a temporary, potentially incomplete query result using all data retrieved so far; from such a result the system reports those solutions that did not appear in the result generated during the previous result computation stage. Hence, the system incrementally explores the queried Web of Data in a breadth-first manner and produces more and more solutions of the query result during that process.

It is easy to see that such a naive, breadth-first LTBQE approach is unsuitable in practice because completely recomputing a partial query result during each result computation stage is not efficient. Schmedding proposes a version of the naive approach that addresses this problem [65]. The idea of Schmedding's approach is to recursively adjust the currently computed query result each time the execution system retrieves additional data. Schmedding's main contribution is an extension of the SPARQL algebra operators that makes the differences between query results computed on different input data explicit; based on the extended algebra, each result computation stage computes the next version of the temporary query result by using only i) the additionally retrieved data and ii) the previously computed version of the result (instead of recomputing everything from scratch as in the naive approach).

By abandoning the idea of using the aforementioned stages as a basis for LTBQE approaches, it becomes possible to achieve an even tighter integration of link traversal and result construction: Instead of performing multiple result computation stages that always compute a whole query result (from scratch as in a the naive approach or incrementally as proposed by Schmedding), Hartig et al. introduce an LTBQE approach that consists of a single result construction process only [36, 33, 37]; this process computes the solutions of a query result by incrementally augmenting intermediate solutions such that these intermediate solutions cover more and more triple patterns of the query. For such an augmentation the process uses matching triples from data retrieved via link traversal. At the same time, the process uses the URIs in these matching triples for further link traversal. Hence, this approach does not follow arbitrary links in the discovered data but only those links that correspond to triple patterns in the executed query.

We emphasize that LTBQE as described by Hartig et al. presents a general strategy rather than a concrete, implementable algorithm. In a more recent publication Hartig and Freytag provide a formal, implementation independent definition of this LTBQE strategy and use this definition to formally analyze the strategy [37]. A variety of approaches for implementing this strategy are conceivable. In the following section we focus on a particular example of these approaches that uses a synchronized pipeline of iterators.

8.4 An Iterator Based Implementation

To build a query execution system that applies the general idea of the LTBQE paradigm, we require a concrete approach for implementing LTBQE. As an example for such an implementation this section discusses an application of the well-known iterator model [25]. We first introduce this implementation approach and describe its characteristics. Then, we discuss the problem of determining query execution plans for this approach and, finally, introduce an optimization that reduces the impact of data access times on the overall query execution times. For the sake of brevity, we only consider SPARQL based Linked Data queries that are expressed using a basic graph pattern (BGP).[8] Since these queries present a form of conjunctive queries, we call them *conjunctive Linked Data queries* (or *CLD queries* for short).

8.4.1 Introduction to the Implementation Approach

A basis for query execution –in general– is a query execution plan that represents the given query as a tree of operations. A well established approach to execute such a plan is *pipelining* in which each solution produced by one operation is passed directly to the operation that uses it [19]. 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 implemented by a tree of *iterator*s, each of which performs a particular operation [25]. 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. 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 computes solutions in a pull fashion.

To use iterators for executing a CLD query in a link traversal based manner we assume a logical execution plan that specifies an order for the triple patterns of the query. Selecting such a plan is an optimization problem that we discuss in Section 8.4.3. The physical implementation of such a logical plan is a sequence of iterators $I_0, I_1, \dots, I_n$ such that the i-th iterator I_i (where $i \in \{1, \dots, n\}$ and n is the number of triple patterns in the query) is responsible for the i-th triple pattern (as given by the selected order). Iterator I_0 is a special iterator; its `Open` function dereferences all seed URIs of the query and uses the retrieved data to initialize a query-local dataset. The `GetNext` function of iterator I_0 provides a single empty solution mapping $\mu_\emptyset$ (i.e., $\mathrm{dom}(\mu_\emptyset) = \emptyset$).

[8]An extension of the presented concepts to support more complex types of SPARQL query patterns is straightforward (as long as the supported queries are monotonic). Alternatively, the solutions for each BGP based subquery that might be computed using LTBQE, may be processed by the SPARQL algebra as usual.

Algorithm 1 Functions of an iterator used for implementing LTBQE.

```
Require: $tp$ – a triple pattern
         $I_{\mathsf{pred}}$ – a predecessor iterator
         $\mathcal{D}$ – the query-local dataset (note, all iterators have access to $\mathcal{D}$)

FUNCTION Open
 1: $I_{\mathsf{pred}}$.Open // initialize the input iterator
 2: $\Omega_{\mathsf{tmp}} := \emptyset$ // for storing (precomputed) solution mappings temporarily

FUNCTION GetNext
 3: while $\Omega_{\mathsf{tmp}} = \emptyset$ do
 4:     $\mu_{\mathsf{input}} := I_{\mathsf{pred}}$.GetNext // consume partial solution from direct predecessor
 5:     if $\mu_{\mathsf{input}}$ = EndOfFile then return EndOfFile end if
 6:     $tp' := \mu_{\mathsf{input}}(tp)$
 7:     Ensure that each URI $u \in \mathrm{uris}(tp')$ has been dereferenced and all re-
        trieved data is available as part of the query-local dataset $\mathcal{D}$.
 8:     $G_{\mathsf{snap}}$ := all RDF triples in the current version of $\mathcal{D}$
 9:     $\Omega_{\mathsf{tmp}} := \{\mu_{\mathsf{input}} \cup \mu' \mid \mathrm{dom}(\mu') = \mathrm{vars}(tp') \text{ and } \mu'(tp') \in G_{\mathsf{snap}}\}$
10: end while
11: $\mu$ := an element in $\Omega_{\mathsf{tmp}}$
12: $\Omega_{\mathsf{tmp}} := \Omega_{\mathsf{tmp}} \setminus \{\mu\}$
13: return $\mu$

FUNCTION Close
14: $I_{\mathsf{pred}}$.Close // close the input iterator
```

We refer to each of the other iterators, I_1 to I_n, as a *link traversing iterator*. Those implement the functions `Open`, `GetNext`, and `Close` as given in Listing 1. We briefly describe the operation executed by the i-th link traversing iterator iterator, I_i (where $i \in \{1, \dots, n\}$). This iterator reports solution mappings that cover the first i triple patterns. To produce these intermediate solutions iterator I_i executes the following four steps repeatedly: First, the iterator consumes a solution mapping μ_{input} from its direct predecessor, iterator I_{i-1}, and applies this mapping to its triple pattern tp_i, resulting in a triple pattern $tp'_i = \mu_{\mathsf{input}}(tp_i)$ (cf. lines 4 to 6 in Listing 1); second, the iterator ensures that the query-local dataset contains all data that can be retrieved by dereferencing all URIs mentioned in tp'_i (cf. line 7); third, the iterator precomputes solution mappings based on all matching triples for tp'_i that are currently available in the query-local dataset (cf. lines 8 and 9); and, fourth, I_i (iteratively) reports each of the precomputed solution mappings (cf. lines 11 to 13).

Example 13. *Let $Q^{B_{\mathsf{ex}}}$ be a CLD query (under full-Web semantics) where $B_{\mathsf{ex}} = \{tp_1, tp_2\}$ is a BGP consisting of triple patterns $tp_1 = (?x,$ `ex:p1`, `ex:a`$)$*

and $tp_2 = (?x, \texttt{ex:p2}, ?y)$. *For a link traversal based execution of this query, we assume a physical plan* I_0, I_1, I_2 *where* I_0 *is the root iterator and link traversing iterators* I_1 *and* I_2 *are responsible for triple patterns* tp_1 *and* tp_2*, respectively. The sequence diagram in Figure 8.1 illustrates an execution of this plan over a Web of Data* $W_{\mathsf{ex}} = (D_{\mathsf{ex}}, data_{\mathsf{ex}}, adoc_{\mathsf{ex}})$ *with:*

$$\begin{aligned}
adoc_{\mathsf{ex}}(\texttt{ex:a}) &= d_{\mathsf{a}}, & data_{\mathsf{ex}}(d_{\mathsf{a}}) &= \big\{(\texttt{ex:b},\ \texttt{ex:p1},\ \texttt{ex:a}),\\
& & &\quad\ (\texttt{ex:c},\ \texttt{ex:p1},\ \texttt{ex:a})\big\},\\
adoc_{\mathsf{ex}}(\texttt{ex:b}) &= d_{\mathsf{b}}, & data_{\mathsf{ex}}(d_{\mathsf{b}}) &= \big\{(\texttt{ex:b},\ \texttt{ex:p2},\ \texttt{ex:d})\big\},\\
adoc_{\mathsf{ex}}(\texttt{ex:c}) &= d_{\mathsf{c}}, & data_{\mathsf{ex}}(d_{\mathsf{c}}) &= \big\{(\texttt{ex:c},\ \texttt{ex:p2},\ \texttt{ex:d})\big\},
\end{aligned}$$

and $\mathrm{dom}(adoc_{\mathsf{ex}}) = \{\, \texttt{ex:a}, \texttt{ex:b}, \texttt{ex:c} \,\}$.

As can be seen from the sequence diagram, the first execution of the ***GetNext*** *function of iterator* I_1 *begins with consuming the empty solution mapping* $\mu_\emptyset$ *from root iterator* I_0. *This solution mapping corresponds to* μ_{input} *in Listing 1 (cf. line 4). Based on* $\mu_\emptyset$, *iterator* I_1 *initializes triple pattern*

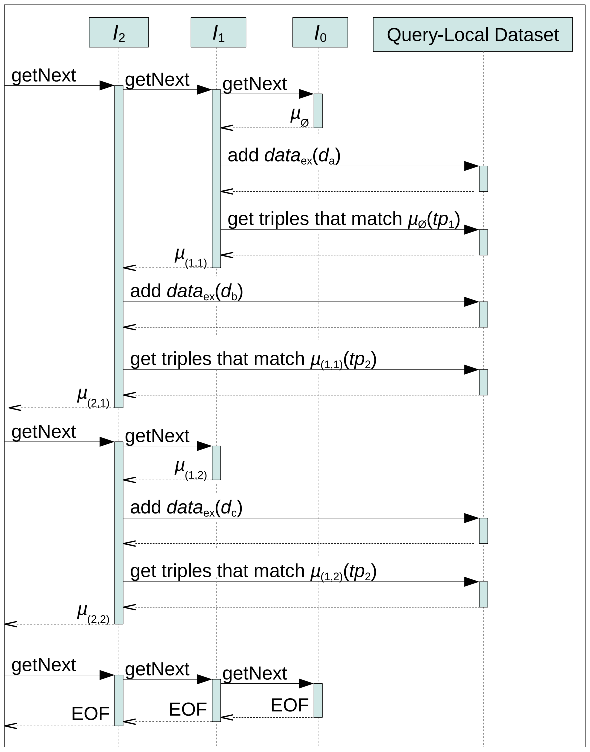

FIGURE 8.1: Sequence diagram that illustrates the interaction between link traversing iterators during a link traversal based query execution process.

$tp'_1 = \mu_\emptyset(tp_1)$ and dereferences all URIs in tp'_1. Note, $tp'_1 = tp_1$ because $\mathrm{dom}(\mu_\emptyset) = \emptyset$*. Thus, I_1 dereferences two URIs,* `ex:p1` *and* `ex:a`*, which, in the case of the queried example Web W_{ex}, results in adding $data_{\mathsf{ex}}(d_{\mathsf{a}})$ to the query-local dataset $\mathcal{D}$. Then, I_1 precomputes a set $\Omega_{\mathsf{tmp}(1)}$ of solutions for triple pattern tp'_1. Since, at this point, the query-local dataset $\mathcal{D}$ contains two RDF triples that match tp'_1, it holds that $\Omega_{\mathsf{tmp}(1)} = \{\mu_{(1,1)}, \mu_{(1,2)}\}$ where $\mu_{(1,1)} = \{?x \rightarrow \texttt{ex:b}\}$, and $\mu_{(1,2)} = \{?x \rightarrow \texttt{ex:c}\}$. After precomputing $\Omega_{\mathsf{tmp}(1)}$, iterator I_1 removes solution mapping $\mu_{(1,1)}$ from this precomputed set and returns that solution mapping as the first result of its operation.*

Using $\mu_{(1,1)}$ as input, iterator I_2 initializes the temporary triple pattern $tp'_2 = \mu_{(1,1)}(tp_2) = (\texttt{ex:b}, \texttt{ex:p2}, ?y)$, dereferences all URIs in tp'_2, and, thus, adds $data_{\mathsf{ex}}(d_{\mathsf{b}})$ to the query-local dataset $\mathcal{D}$. Thereafter, I_2 precomputes (a first version of) its set Ω_{tmp} for tp'_2. To denote this particular version of Ω_{tmp} we write $\Omega_{(2,1)}$. Since one triple in the query-local dataset matches tp'_2 (namely, the triple that comes from LD document d_{b}), we have $\Omega_{(2,1)} = \{\mu_{(2,1)}\}$ with $\mu_{(2,1)} = \{?x \rightarrow \texttt{ex:b}, ?y \rightarrow \texttt{ex:d}\}$. It is easy to see that $\mu_{(2,1)}$ is a solution for BGP B_{ex}. Iterator I_2 concludes the first execution of its ***GetNext*** *function by reporting solution $\mu_{(2,1)}$, after removing this solution from Ω_{tmp}. As a consequence, Ω_{tmp} is empty when the query execution system requests another solution from I_2 (by calling the* ***GetNext*** *function of I_2 a second time). Hence, at the begin of the second execution of* ***GetNext****, I_2 consumes the next solution mapping from its predecessor I_1.*

In the remaining steps of the query execution, the iterators proceed as illustrated in Figure 8.1 (where $\mu_{(2,2)} = \{?x \rightarrow \texttt{ex:c}, ?y \rightarrow \texttt{ex:d}\}$).

Notice, in contrast to iterators usually used for computing queries over a fixed dataset, calling a link traversing iterator may have the side-effect of an augmentation of the query-local dataset (as desired for an implementation of the LTBQE paradigm). In particular, link traversing iterators augment the query-local dataset based on the following *solution discoverability assumption*: RDF triples that match a triple pattern are most likely to find in the data that can be retrieved by dereferencing the URIs mentioned in the triple pattern. Hence, the query-local availability of this data may improve chances to increase the number of query solutions that can be reported. Such an assumption is justified by the common practice of publishing Linked Data.

8.4.2 Characteristics of the Implementation Approach

We now discuss characteristics and limitations that are specific to the presented approach of implementing LTBQE using a pipeline of iterators.

Most importantly, the implementation approach is sound, that is, any solution mapping reported by the last iterator in the pipeline is in fact a solution of the corresponding query result as specified in Definition 17 [37].[9]

[9]In fact, Hartig and Freytag show that the implementation approach is sound for SPARQL based Linked Data queries under a more restrictive, reachability based query

However, the approach cannot guarantee to compute all solutions that may be computed using the final snapshot of the query-local dataset. As the primary reason for this limitation we note that each link traversing iterator discards an input solution mapping after using it. More precisely, any link traversing iterator I_i uses each input solution mapping μ_{input} only once to find matching triples for triple pattern $tp'_i = \mu_{\mathsf{input}}(tp_i)$ (where tp_i is the triple pattern that iterator I_i is responsible for). As a consequence, iterator misses those matching triples for tp'_i that any of the iterators discovers and adds to the query-local dataset after I_i made use of intermediate solution μ_{input}.

Discarding intermediate solutions after using them may additionally cause another, somewhat unexpected effect. That is, the order in which the triple patterns of a query are assigned to the iterators (i.e., the logical query plan) may influence which LD documents the iterators discover and, thus, which solutions they report. The following example illustrates this effect:

```
?x rdf:type ex:X .
?x ex:p1 ?y .
?y rdfs:label ?z .
?y ex:p2 ex:a .
```

Example 14. *To execute a CLD query $\mathcal{Q}^{B_{\mathsf{ex}}}$ where B_{ex} is the BGP in the adjoining figure we may select a query plan that orders the triple patterns in the same order in which they are listed in the figure. During the execution of this plan, iterator I_2 requests a first intermediate solution from its predecessor I_1. Iterator I_1, which is responsible for the* `rdf:type` *triple pattern, ensures that the query-local dataset contains the data that can be retrieved by dereferencing URIs* `rdf:type` *and* `ex:X`*. For the sake of the example we assume that the queried Web is a Web of Data $W_{\mathsf{ex}} = (D_{\mathsf{ex}}, data_{\mathsf{ex}}, adoc_{\mathsf{ex}})$ such that:*

$$\begin{aligned}
adoc_{\mathsf{ex}}(\texttt{ex:X}) &= d_{\texttt{X}}, & data_{\mathsf{ex}}(d_{\texttt{X}}) &= \big\{(\texttt{ex:X},\ \texttt{rdfs:subClassOf},\ \texttt{ex:Y})\big\},\\
adoc_{\mathsf{ex}}(\texttt{ex:a}) &= d_{\mathsf{a}}, & data_{\mathsf{ex}}(d_{\mathsf{a}}) &= \big\{(\texttt{ex:b},\ \texttt{ex:p2},\ \texttt{ex:a})\big\},\\
adoc_{\mathsf{ex}}(\texttt{ex:b}) &= d_{\mathsf{b}}, & data_{\mathsf{ex}}(d_{\mathsf{b}}) &= \big\{(\texttt{ex:b},\ \texttt{rdfs:label},\ \texttt{"..."}),\\
&& &\qquad (\texttt{ex:c},\ \texttt{ex:p1},\ \texttt{ex:b})\big\},\\
adoc_{\mathsf{ex}}(\texttt{ex:c}) &= d_{\mathsf{c}}, & data_{\mathsf{ex}}(d_{\mathsf{c}}) &= \big\{(\texttt{ex:c},\ \texttt{rdf:type},\ \texttt{ex:X})\big\},
\end{aligned}$$

and $\mathrm{dom}(adoc_{\mathsf{ex}}) = \{\, \texttt{ex:X}, \texttt{ex:a}, \texttt{ex:b}, \texttt{ex:c} \,\}$*. Hence, when iterator I_1 tries to find matching triples for its triple pattern, the query-local dataset includes only $data_{\mathsf{ex}}(d_{\texttt{X}})$ (assuming that no other seed URIs have been specified for the query). Apparently, $data_{\mathsf{ex}}(d_{\texttt{X}})$ does not contain triples that match I_1's triple pattern. Therefore, I_1 cannot generate and report any intermediate solution and, thus, the overall query result as computed based on the selected plan is empty. Even if we assume a set of seed URIs that consists of all URIs mentioned in BGP B_{ex}, iterator I_1 could still not find a matching triple for the first triple pattern. However, an alternative query plan may use the reverse order (i.e., in this plan iterator I_1 would be responsible for the* `ex:p2` *triple*

semantics [37, Theorem 5]. Soundness for full-Web semantics follows trivially because any query result under the reachability based semantics considered by Hartig and Freytag is a subset of the corresponding query results under full-Web semantics [34, Proposition 3].

pattern). By executing this plan we would obtain a solution given as follows: $\mu = \{(?\texttt{x}, \texttt{ex:c}), (?\texttt{y}, \texttt{ex:b}), (?\texttt{z}, \texttt{"..."})\}$.

As can be seen from the example, the iterator based implementation of LTBQE may return different result sets for a CLD query depending on the evaluation order selected for the triple patterns of the query. Such a behavior can be explained based on the following two orthogonal phenomena:

Missing backlinks: On the traditional, hypertext Web it is unusual that Web pages are linked bidirectionally. Similarly, for a Web of Data $W = (D, data, adoc)$ an RDF triple of the form $(s, p, o) \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$ contained in $data(adoc(s))$ (respectively in $data(adoc(o))$) does not have to be contained in $data(adoc(o))$ (respectively in $data(adoc(s))$). We speak of a *missing backlink*. Due to missing backlinks it is possible that one evaluation order allows for the discovery of a matching triple whereas another order misses that triple. For instance, the reason for the different results in Example 14 is a missing backlink in $data_{\mathsf{ex}}(d_{\texttt{X}})$.

Serendipitous discovery: Based on the aforementioned solution discoverability assumption any link traversing iterator I_i enforces the dereferencing of certain URIs because the corresponding data may contain matching triples for the triple pattern tp'_i currently evaluated by the iterator. However, even if dereferenced for the evaluation of a specific triple pattern, the data retrieved using such a URI $u^* \in \text{uris}(tp'_i)$ may also contain an RDF triple t^* that matches another triple pattern tp'_j which will be evaluated later by any of the iterators. Let us assume that i) $u^* \notin \text{uris}(tp'_j)$ and ii) $t^* \notin data(adoc(u))$ for all URIs $u \in \text{uris}(tp'_j)$. Then, RDF triple t^* cannot be discovered by performing line 7 in Listing 1 for triple pattern tp'_j. However, since t^* has been discovered before, it can be used for generating an additional solution mapping during the evaluation of tp'_j. We say that this solution mapping has been *discovered by serendipity*. If the triple patterns of the query were ordered differently, t^* may not be discovered before the evaluation of tp'_j an, thus, the serendipitously discovered solution mapping may not be generated.

The dependency of result completeness on the order in which the triple patterns of a CLD query are evaluated by link traversing iterators implies that certain orders are more suitable than others. In the following we discuss the problem of selecting a suitable order.

8.4.3 Selecting Query Plans for the Approach

Selecting a logical query execution plan that specifies an order for the triple patterns in a given CLD query is an optimization problem: Different plans for the same query may exhibit different performance characteristics. These differences may not only affect query execution times (and other, resource requirements oriented measures) but also result completeness, as we have seen in the previous section. Consequently, we are interested in selecting a logical query execution plan that has low *cost* (e.g., measured in terms of the overall time for execution) as well as high *benefit*, measured as the number of solutions that an execution of the plan returns.[10]

We emphasize that there is an inherent trade-off between cost and benefit: A plan that is able to report more solutions may require more resources than a plan that reports less solutions. In particular, such a highly beneficial plan is likely to retrieve more data which would occupy more query-local memory and would increase query execution times.

To rank and select query plans it is necessary to calculate (or estimate) their cost and their benefit without executing them. Such a calculation requires information about reachable data and the data sources involved in the execution of a plan. However, such information is not available when we start a link traversal based query execution. We just have a query and an empty query-local dataset; we do not know anything about the LD documents we will discover; we do not even know what LD documents will be discovered. As a consequence of this complete lack of information the application of a cost (and benefit) based ranking of plans is unsuitable in our scenario.

As an alternative we may use a heuristics based approach for plan selection. In the following we specify four heuristic rules that may be used for such a purpose. We also describe the rationale for each of these rules.

1. The DEPENDENCY RULE proposes to use a *dependency respecting query plan*, that is, an order for the BGP of a given CLD query such that at least one of the query variables in each triple pattern of the BGP occurs in one of the preceding triple patterns. Formally, an ordered BGP[11] $\bar{B} = [tp_1, \ldots, tp_n]$ is dependency respecting if for each $i \in \{2, \ldots, n\}$ there exist a $j < i$ and a query variable $?v \in \mathrm{vars}(tp_i)$ such that $?v \in \mathrm{vars}(tp_j)$. It is easy to see that for each BGP that represents a connected graph[12] it is always possible to find a dependency respecting query plan. For BGPs that do not represent a connected graph (which are rarely used in practice) at least each connected component should be ordered according to the DEPENDENCY RULE.

   Rationale: Dependency respecting query plans are a reasonable requirement because those enable each iterator to reuse some of the bindings in each input solution mapping μ_{input} consumed from their predecessor iterator. This strategy avoids what can be understood to be an equivalent to the calculation of cartesian products in RDBMS query executions.

2. The SEED RULE proposes to use a plan in which the first triple pattern contains as many HTTP URIs as possible. Formally, for the triple

[10]Another important dimension that we do not account for here is response time, that is, query executions should report first solutions as early as possible.

[11]We represent *ordered BGPs* as lists, denoted by comma-separated elements in brackets.

[12]A BGP B represents a connected graph if for each pair of BGPs B_1 and B_2 with $B_1 \cup B_2 = B$ and $B_1 \cap B_2 = \emptyset$ it holds that $\exists tp_i \in B_1, tp_j \in B_2 : \mathrm{vars}(tp_i) \cap \mathrm{vars}(tp_j) \neq \emptyset$.

pattern $tp_1 \in B$ selected as first triple pattern in a plan for a CLD query with BGP B it must hold that there does not exist another triple pattern $tp_i \in B \setminus \{tp_1\}$ such that $|\mathrm{vars}(tp_1)| < |\mathrm{vars}(tp_i)|$.

Rationale: During any query execution of a sequence $I_0, I_1, \ldots, I_n$ of iterators, iterator I_1 (which is responsible for the first triple pattern in the query) is the first iterator that generates solution mappings based on matching triples in the query-local dataset. The remaining link traversing iterators augment these solution mappings by adding bindings for their query variables. Hence, we need to select a triple pattern for I_1 such that there is a high likelihood that the early snapshot of the query-local dataset as used by I_1 already contains matching triples for the selected, first triple pattern. According to the aforementioned solution discoverability assumption (cf. Section 8.4.1), matching triples for a triple pattern might be found, in particular, in RDF graphs that can be retrieved by looking up the URIs that are part of this pattern. Therefore, it is reasonable to select one of triple patterns with the maximum number of mentioned HTTP URIs as the first triple pattern.

3. The Instance Seed Rule proposes to avoid query plans in which the first triple pattern contains only URIs that denote vocabulary terms. Such a triple pattern can be identified with high likelihood by a simple syntactical analysis: Since URIs in the predicate position always denote vocabulary terms, a preferred first triple pattern must contain an HTTP URI in subject or object position. However, in triple patterns with a predicate of `rdf:type` a URI in the object position identifies a class, i.e., also a vocabulary term. Hence, these triple patterns should also be avoided as first triple pattern.

   Rationale: By narrowing down the set of candidate query plans using the Instance Seed Rule we can expect to increase the average benefit of the remaining set of plans. This expectation is based on the following observation: URIs which identify vocabulary terms resolve to RDF data that usually contains vocabulary definitions and very little or no instance data. However, queries usually ask for instance data and do not contain patterns that have to match vocabulary definitions. Hence, it is reasonable to avoid a first triple pattern whose URIs are unlikely to yield instance data as a starting point for query execution. Example 14 illustrates the negative consequences of ignoring the Instance Seed Rule by selecting the `rdf:type` triple pattern as the first triple pattern for the query plan. Notice, for applications that mainly query for vocabulary definitions the rule must be adjusted.

4. The Filter Rule proposes to prefer query plans in which *filtering triple patterns* are placed as close to the first triple pattern as possible. A filtering triple pattern in an ordered BGP contains only query variables that are also contained in at least one preceding triple pattern. Formally,

a triple pattern tp_i in an ordered BGP $\bar{B} = [tp_1, \dots, tp_n]$ is a filtering triple pattern if for each variable $?v \in \text{vars}(tp_i)$ there exists a $j < i$ such that $?v \in \text{vars}(tp_j)$. Additionally, any triple pattern tp with $\text{vars}(tp) = \emptyset$ is trivially a filtering triple pattern.

Rationale: The rationale of the FILTER RULE is to reduce the number of irrelevant intermediate solutions as early as possible and, thus, to ultimately reduce query execution times: During query execution, each input solution mapping μ_{input} consumed by an iterator I_i is guaranteed to contain bindings for *all* variables in I_i's triple pattern tp_i if and only if tp_i is a filtering triple pattern. Therefore, the application of these input solution mappings to pattern tp_i (cf. line 6 in Listing 1) will always result in a triple pattern without variables, i.e., an RDF triple. If this triple is contained in the query-local dataset, the iterator simply passes on the current μ_{input}; otherwise, it discards this intermediate solution. Thus, any iterator that evaluates a filtering triple patterns may reduce the number of intermediate solutions but it will never multiply this number; i.e., it will not report more solution mappings than it consumes). For iterators whose triple patterns are not filtering triple patterns such a behavior cannot be predicted, neither a reduction nor a multiplication of intermediate solutions.

We emphasize that the FILTER RULE might not always be beneficial because it reduces the likelihood for serendipitous discovery that we discussed in Section 8.4.2. However, during an experimental evaluation of the heuristic such a hypothetical reduction of benefit did not occur in practice [33].

For our scenario in which we cannot assume any information about statistics or data distribution when we start the execution of a query, using heuristics, such as the ones introduced, is a suitable strategy for plan selection. However, we note that after starting the query execution it becomes possible to gather information and observe the behavior of the selected plan. This may allow the query system to reassess candidate plans and, thus, to adapt or even replace the running plan. To the best of our knowledge, such a strategy of adaptive query planning has not yet been studied in the context of LTBQE (nor for any other Linked Data query execution approach).

8.4.4 Improving Performance of the Approach

Independent of the query plan that has been selected to execute a CLD query using a pipeline link traversing iterators, the URI lookups necessary for augmenting the query-local dataset may require a significant amount of time due to network latencies, etc. As a consequence, even if link traversing iterators support a pipelined execution (i.e., complete materialization of intermediate results is not necessary), the execution may block temporarily when an iterator waits for the completion of certain URI lookups. In this section we conclude our discussion of the iterator implementation of LTBQE by outlining some strategies on how to avoid such a temporary blocking and to reduce the impact of network access times on the overall query execution time.

The dereferencing of URIs in line 7 of Listing 1 should be implemented by asynchronous function calls such that multiple dereferencing tasks can be processed in parallel. However, waiting for the completion of these dereferencing tasks in line 7 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 a URI at the time when the corresponding data is required (i.e., at line 7), the dereferencing task can be *initiated* as soon as the URI becomes part of a precomputed solution mapping (i.e., at line 9). Then, the query engine can immediately proceed the evaluation while the asynchronous 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. Experiments show that this prefetching strategy reduces query execution times to about 80% [36].

Further improvements can be achieved with an optimization approach presented by Hartig et al. [36]. The authors study the possibility to postpone processing of intermediate solutions for which necessary URI lookups are still pending. To conceptually integrate this idea into the iterator based implementation approach Hartig et al. propose to extend the iterator model with a fourth function, called `Postpone`. The general semantics of this `Postpone` function is to treat the element most recently reported by the `GetNext` function as if this element has not yet been reported. Hence, `Postpone` "takes back" this element and `GetNext` may report it (again) in response to a later request. In the particular case of link traversing iterators, these elements are the solution mappings computed by the iterators. Thus, link traversing iterators may use the proposed extension to temporarily ignore those input solution mappings whose processing would cause blocking. This approach results in reducing the overall query execution times to about 50% [36].

8.4.5 Alternative Implementation Approaches

We have already mentioned that the general LTBQE strategy introduced by Hartig et al. [36, 37] can be implemented in multiple ways. Hence, using iterators as presented in this chapter is only one possible implementation approach. Other implementations studied so far can be summarized as follows:

- Ladwig and Tran propose an implementation that uses symmetric hash join operators which are connected via an asynchronous, push-based pipeline [49]. In later work, the authors extend this approach and introduce the *symmetric index hash join* operator. This operator allows a query execution system to incorporate a locally existing RDF data set into the query execution [50].

- Miranker et al. introduce another push-based implementation [57]. The authors implement LTBQE using the well-known Rete match algorithm.

8.5 Summary

In this chapter we have introduced link traversal based query execution (LTBQE), a novel query execution paradigm that is tailored to query Linked Data on the Web. The most important feature of LTBQE is the possibility to use data from initially unknown data sources. This possibility allows for serendipitous discovery and, thus, enables applications that tap the full potential of the Web of Data. Another feature is that an LTBQE system does not require any a-priori information about the queried Web. As a consequence, such a system can directly be used without having to wait for the completion of an initial data load phase or any other type of preprocessing. Hence, LTBQE is most suitable for an "on-demand" live querying scenario where freshness and discovery of results is more important than an almost instant answer.

Bibliography

[1] Ben Adida, Mark Birbeck, Shane McCarron, and Ivan Herman. RDFa Core 1.1. W3C Recommendation, June 2012. `http://www.w3.org/TR/rdfa-syntax/`.

[2] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. RDFa in XHTML: Syntax and Processing. W3C Recommendation, October 2008. `http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014/`.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Application*. Cambridge University Press, 2002.

[4] Dave Beckett. RDF/XML Syntax Specification (Revised). W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[5] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Turtle: Terse RDF Triple Language. W3C Working Draft, July 2012. `http://www.w3.org/TR/turtle/`.

[6] Tim Berners-Lee. Semantic Web Road map. W3C Design Issues, September 1998. `http://www.w3.org/DesignIssues/Semantic.html`.

[7] Tim Berners-Lee. Linked Data. W3C Design Issues, July 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[8] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. W3C Team Submission, March 2011. `http://www.w3.org/TeamSubmission/n3/`.

[9] Tim Berners-Lee, Jim Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 5(284):35–40, 2001.

[10] Diego Berrueta and Jon Phipps. Best Practice Recipes for Publishing RDF Vocabularies. W3C Working Group Note, August 2008. `http://www.w3.org/TR/swbp-vocab-pub/`.

[11] Mark Birbeck and Shane McCarron. CURIE Syntax 1.0: A syntax for expressing Compact URIs. W3C Working Group Note, December 2010. `http://www.w3.org/TR/curie/`.

[12] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia – a crystallization point for the Web of Data. *J. Web Sem.*, 7(3):154–165, 2009.

[13] Paolo Bouquet, Chiara Ghidini, and Luciano Serafini. Querying The Web Of Data: A Formal Approach. In *Proc. of the 4th Asian Semantic Web Conference (ASWC)*, 2009.

[14] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-schema/`.

[15] Dan Brickley, R.V. Guha, and Andrew Layman. Resource Description Framework (RDF) Schemas. W3C Recommendation, April 1998. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[16] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. SPARQL Query Language for RDF. W3C Recommendation, January 2008. `http://www.w3.org/TR/rdf-sparql-protocol/`.

[17] Li Ding and Tim Finin. Characterizing the Semantic Web on the Web. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2006.

[18] Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Peter F. Patel-Schneider. OIL: An ontology infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[19] Hector Garcia-Molina, Jennifer Widom, and Jeffrey D. Ullman. *Database Systems: The Complete Book*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[20] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 Federated Query. W3C Proposed Recommendation, November 2012. `http://www.w3.org/TR/sparql11-update/`.

[21] Birte Glimm, Aidan Hogan, Markus Krötzsch, and Axel Polleres. OWL: Yet to arrive on the Web of Data? In Christian Bizer, Tom Heath, Tim Berners-Lee, and Michael Hausenblas, editors, *LDOW*, volume 937 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[22] Birte Glimm and Chimezie Ogbuji. SPARQL 1.1 Entailment Regimes. W3C Candidate Recommendation, November 2012. `http://www.w3.org/TR/sparql11-entailment/`.

[23] Birte Glimm and Sebastian Rudolph. Status QIO: Conjunctive query entailment is decidable. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *KR*. AAAI Press, 2010.

[24] Christine Golbreich and Evan K. Wallace. OWL 2 Web Ontology Language: New Features and Rationale. W3C Recommendation, October 2009. `http://www.w3.org/TR/owl2-new-features/`.

[25] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[26] Jan Grant and Dave Beckett. RDF Test Cases. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-testcases/`.

[27] Bernardo Cuenca Grau, Boris Motik, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language: Profiles. W3C Recommendation, October 2009. `http://www.w3.org/TR/owl2-profiles/`.

[28] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *WWW*, pages 48–57, 2003.

[29] Harry Halpin, Patrick J. Hayes, James P. McCusker, Deborah L. McGuinness, and Henry S. Thompson. When owl:sameAs isn't the same: An analysis of identity in Linked Data. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *International Semantic Web Conference (1)*, volume 6496 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2010.

[30] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Proposed Recommendation, November 2012. `http://www.w3.org/TR/sparql11-query/`.

[31] Andreas Harth and Sebastian Speiser. On Completeness Classes for Query Evaluation on Linked Data. In *Proc. of the 26th AAAI Conference*, 2012.

[32] Olaf Hartig. How Caching Improves Efficiency and Result Completeness for Querying Linked Data. In *Proc. of the 4th Linked Data on the Web workshop (LDOW) at WWW*, 2011.

[33] Olaf Hartig. Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution. In *Proc. of the 8th Extended Semantic Web Conference (ESWC)*, 2011.

[34] Olaf Hartig. SPARQL for a Web of Linked Data: Semantics and Computability. In *Proc. of the 9th Extended Semantic Web Conference (ESWC)*, 2012.

[35] Olaf Hartig. An Overview on Execution Strategies for Linked Data Queries. *Datenbank-Spektrum*, 13(2), 2013.

[36] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In *Proc. of the 8th International Semantic Web Conference (ISWC)*, 2009.

[37] Olaf Hartig and Johann-Christoph Freytag. Foundations of Traversal Based Query Execution over Linked Data. In *Proc. of the 23rd ACM Conference on Hypertext and Social Media (HT)*, 2012.

[38] Jonathan Hayes and Claudio Gutiérrez. Bipartite graphs as intermediate model for RDF. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2004.

[39] Patrick Hayes. RDF Semantics. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-mt/`.

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

[41] James Hendler and Deborah L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):67–73, 2000.

[42] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, May 2004. `http://www.w3.org/Submission/SWRL/`.

[43] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. Reviewing the design of DAML+OIL: An ontology language for the Semantic Web. In *AAAI/IAAI*, pages 792–797, 2002.

[44] Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation, December 2004. `http://www.w3.org/TR/webarch/`.

[45] Ilianna Kollia, Birte Glimm, and Ian Horrocks. SPARQL query answering over OWL ontologies. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *ESWC (1)*, volume 6643 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2011.

[46] Markus Krötzsch, Frederick Maier, Adila Krisnadhi, and Pascal Hitzler. A better uncle for OWL: nominal schemas for integrating rules and ontologies. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *WWW*, pages 645–654. ACM, 2011.

[47] Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Description Logic Rules. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 80–84. IOS Press, 2008.

[48] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A Description Logic Primer. *CoRR*, abs/1201.4089, 2012.

[49] Günter Ladwig and Duc Thanh Tran. Linked Data query processing strategies. In *Proc. of the 9th International Semantic Web Conference (ISWC)*, 2010.

[50] Günter Ladwig and Duc Thanh Tran. SIHJoin: Querying Remote and Local Linked Data. In *Proc. of the 8th Extended Semantic Web Conference (ESWC)*, 2011.

[51] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, February 1999. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[52] Sean Luke, Lee Spector, David Rager, and James A. Hendler. Ontology-based Web Agents. In *Agents*, pages 59–66, 1997.

[53] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On blank nodes. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7031 of *Lecture Notes in Computer Science*, pages 421–437. Springer, 2011.

[54] Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-primer/`.

[55] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, February 2004. `http://www.w3.org/TR/owl-features/`.

[56] Alistair Miles, Thomas Baker, and Ralph Swick. Best Practice Recipes for Publishing RDF Vocabularies. W3C Working Draft, March 2006. `http://www.w3.org/TR/2006/WD-swbp-vocab-pub-20060314/` (Later superseded by [10]).

[57] Daniel P. Miranker, Rodolfo K. Depena, Hyunjoon Jung, Juan F. Sequeda, and Carlos Reyna. Diamond: A SPARQL Query Engine, for Linked Data Based on the Rete Match. In *Proc. of the Workshop on Artificial Intelligence meets the Web of Data (AImWD) at ECAI*, 2012.

[58] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Simple and efficient minimal RDFS. *J. Web Sem.*, 7(3):220–234, 2009.

[59] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[60] Axel Polleres. From SPARQL to rules (and back). In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 787–796. ACM, 2007.

[61] Axel Polleres. How (well) do Datalog, SPARQL and RIF interplay? In Pablo Barceló and Reinhard Pichler, editors, *Datalog*, volume 7494 of *Lecture Notes in Computer Science*, pages 27–30. Springer, 2012.

[62] Eric Prud'hommeaux and Carlos Buil-Aranda. SPARQL 1.1 Federated Query. W3C Proposed Recommendation, November 2012. `http://www.w3.org/TR/sparql11-federated-query/`.

[63] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[64] Sebastian Rudolph. Foundations of Description Logics. In Axel Polleres, Claudia d'Amato, Marcelo Arenas, Siegfried Handschuh, Paula Kroner, Sascha Ossowski, and Peter F. Patel-Schneider, editors, *Reasoning Web*, volume 6848 of *Lecture Notes in Computer Science*, pages 76–136. Springer, 2011.

[65] Florian Schmedding. Incremental SPARQL Evaluation for Query Answering on Linked Data. In *Proc. of the 2nd Int. Workshop on Consuming Linked Data (COLD) at ISWC*, 2011.

[66] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

[67] Michael Schneider. OWL 2 Web Ontology Language RDF-Based Semantics. W3C Recommendation, October 2009. `http://www.w3.org/TR/owl2-rdf-based-semantics/`.

[68] Andy Seaborne. SPARQL 1.1 Query Results CSV and TSV Formats. W3C Proposed Recommendation, November 2012. `http://www.w3.org/TR/sparql11-results-csv-tsv/`.

[69] Andy Seaborne. SPARQL 1.1 Query Results JSON Format. W3C Proposed Recommendation, November 2012. `http://www.w3.org/TR/sparql11-results-json/`.

[70] Manu Sporny. JSON-LD Syntax 1.0. W3C Working Draft, July 2012. `http://www.w3.org/TR/json-ld-syntax/`.

[71] Patrick Stickler. CBD – Concise Bounded Description. W3C Recommendation, June 2005. `http://www.w3.org/Submission/CBD/`.

[72] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *J. Web Sem.*, 3(2–3):79–115, 2005.

[73] Denny Vrandečíc, Markus Krötzsch, Sebastian Rudolph, and Uta Lösch. Leveraging non-lexical knowledge for the Linked Open Data Web. *Review of April Fool's day Transactions (RAFT)*, 5:18–27, 2010.

[74] Gregory Todd Williams. SPARQL 1.1 Service Description. W3C Proposed Recommendation, November 2012. `http://www.w3.org/TR/sparql11-service-description/`.

[75] David Wood, Stefan Decker, and Ivan Herman, editors. *Proceedings of the W3C Workshop – RDF Next Steps, Stanford, Palo Alto, CA, USA, June 26–27*. Online at `http://www.w3.org/2009/12/rdf-ws/`, 2010.