Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution

Olaf Hartig

Humboldt-Universität zu Berlin
hartig@informatik.hu-berlin.de

Abstract. Link traversal based query execution is a new query execution paradigm for the Web of Data. This approach allows the execution engine to discover potentially relevant data during the query execution and, thus, enables users to tap the full potential of the Web. In earlier work we propose to implement the idea of link traversal based query execution using a synchronous pipeline of iterators. While this idea allows for an easy and efficient implementation, it introduces restrictions that cause less comprehensive result sets. In this paper we address this limitation. We analyze the restrictions and discuss how the evaluation order of a query may affect result set size and query execution costs. To identify a suitable order, we propose a heuristic for our scenario where no a-priory information about relevant data sources is present. We evaluate this heuristic by executing real-world queries over the Web of Data.

1 Introduction

While the possibility to query the emerging Web of Data enables exciting opportunities, executing SPARQL queries over the Web poses novel challenges [1]. It is impossible to know all data sources that might contribute to the answer of a query. To tap the full potential of the Web, traditional query execution paradigms are insufficient because they assume knowledge of a fixed set of potentially relevant data sources beforehand. In [2] we propose a novel query execution paradigm that conceives the Web of Data as an initially unknown set of data sources and makes use of the characteristics of Linked Data, in particular, the existence of links between data items from different sources. The main idea of our approach is to intertwine the construction of the query result with the traversal of data links that correspond to intermediate solutions in the construction process. This strategy, which we call *link traversal based query execution*, allows the execution engine to discover potentially relevant data during the query execution.

Different implementations of the general idea of link traversal based query execution are possible (e.g. [2,3]), each having its own strengths and drawbacks. In [2] we propose an iterator based implementation approach, including concepts that improve its execution times. This implementation approach applies a synchronized pipeline of operators that evaluate the query in a fixed order. While

this approach determines query results efficiently, the fixed evaluation order may prevent finding some query results. In this paper we address this limitation.

As a prerequisite to analyze the iterator based approach we provide a definition of link traversal based query execution, independent of possible implementation approaches, and, thereby, introduce a completeness criteria. We align our iterator based approach with this definition: We prove that the approach is sound and analyze why it cannot guarantee results that satisfy our completeness criteria. Furthermore, we describe how the evaluation order of the query may affect result completeness. Since this effect causes the need to select a suitable order, we discuss the possibilities of query planning and propose a heuristics based approach as the only applicable strategy in our scenario in which we cannot assume any information about statistics or data distribution when we start the execution of a query. To evaluate our heuristic we execute real-world queries.

This paper is structured as follows: While Section 2 defines link traversal based query execution, Section 3 aligns our implementation approach with this definition and analyzes the issue of result completeness. Section 4 discusses query planning and our heuristic for plan selection. In Section 5 we evaluate this heuristic. Finally, we study related work in Section 6 and conclude in Section 7.

2 Definition of Link Traversal Based Query Execution

Link traversal based query execution is a new query execution paradigm developed to exploit the Web of Data to its full potential. Since adhering to the Linked Data principles is the minimal requirement for publication in the Web of Data our approach relies solely on these principles instead of assuming the existence of source-specific query services such as SPARQL endpoints. This section provides a formal definition of the general idea of link traversal based query execution. For the formalization we adopt a static view of the Web, that is, we assume no changes are made to the data on the Web during the execution of a query.

2.1 Preliminaries

The Linked Data principles require to describe data using RDF. RDF distinguishes three distinct sets of *RDF terms*: U, the (possibly infinite) set of URIs, L, an infinite set of literals, and B, an infinite set of blank nodes that represent unnamed entities. An *RDF triple* is a 3-tuple $t = (s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ where s is called the *subject* of t, p the *predicate*, and o the *object*.

In the Web of Data entities have to be identified via HTTP scheme based URIs. Let $U^{\mathrm{LD}} \subset U$ be the (possibly infinite) set of all these URIs. By looking up such a URI we retrieve RDF data about the entity identified by the URI. For our formalization we introduce a function, denoted as *lookup*, to refer to the result of such look-ups: *lookup* is a surjective function which returns for each URI $u \in U^{\mathrm{LD}}$ a *descriptor object*, that is, a set of RDF triples which i) can be retrieved by looking up u on the Web and which ii) describes the entity identified by u. Hence, based on the Linked Data principles we expect:

$\forall u \in U^{\mathrm{LD}} : \big(\exists (s,p,o) \in \mathit{lookup}(u) : s = u \vee o = u\big)$. Note, *lookup* is not injective; it is possible that the same descriptor object is retrieved by looking up distinct URIs. In this case, the descriptor object describes multiple entities. For each $t \notin U^{\mathrm{LD}}$ the look-up function returns an empty descriptor object: $\mathit{lookup}(t) = \varnothing$.

We define our query execution approach for basic graph patterns[1]. A *basic graph pattern* (BGP) is a subset of the set[2] $(U \cup V) \times (U \cup V) \times (U \cup V \cup L)$ where V is an infinite set of query variables. The elements of a BGP are called *triple patterns*. For each triple pattern tp we write $\mathit{uris}(tp)$ and $\mathit{vars}(tp)$ to denote the set of all URIs and the set of all query variables contained in tp, respectively. A *matching triple* in a set of RDF triples G for a triple pattern $(\tilde{s}, \tilde{p}, \tilde{o})$ is any RDF triple $(s,p,o) \in G$ with $(\tilde{s} \notin V \Rightarrow \tilde{s} = s) \wedge (\tilde{p} \notin V \Rightarrow \tilde{p} = p) \wedge (\tilde{o} \notin V \Rightarrow \tilde{o} = o)$.

2.2 Link Traversal Based Solutions for Basic Graph Patterns

We define the notion of solutions for the link traversal based query execution of a BGP using a two-phase approach: First, we define what descriptor objects can be discovered during link traversal based query execution. Then, we formalize solutions as sets of variable bindings that correspond to a subset of all data from all discovered descriptor objects. Notice, while this two-phase approach provides for a straightforward definition of solutions it does not correspond to the actual query execution strategy of intertwining the traversal of data links and graph pattern matching as is characteristic for link traversal based query execution.

To formalize what descriptor objects can be discovered during the link traversal based execution of a BGP we introduce the concept of *reachability*:

Definition 1. *Let $b = \{tp_1, \ldots, tp_n\}$ be a BGP; let D be a descriptor object. D is* reachable *by the execution of b iff either*

- $\exists (\tilde{s}, \tilde{p}, \tilde{o}) \in b : \mathit{lookup}(\tilde{s}) = D \vee \mathit{lookup}(\tilde{p}) = D \vee \mathit{lookup}(\tilde{o}) = D$
- *or there exists another descriptor object D', a triple pattern $tp \in b$, and an RDF triple $t = (s,p,o)$ such that i) D' is reachable by the execution of b, ii) t is a matching triple for tp in D', and iii) $\mathit{lookup}(s) = D$, $\mathit{lookup}(p) = D$ or $\mathit{lookup}(o) = D$.*

To represent the solutions of BGPs we adopt the notion of a *solution mapping* as defined in the SPARQL specification [4]. These mappings bind query variables to RDF terms. Hence, a solution mapping μ is a set of variable-term-pairs where no two pairs contain the same variable. The application of a solution mapping μ to a triple pattern tp, denoted as $\mu[tp]$, implies replacing each variable in tp by the RDF term it is bound to in μ; unbound variables must not be replaced. Similarly, a solution mapping μ can be applied to a whole BGP b: $\mu[b] = \{\mu[tp_i] \mid tp_i \in b\}$. Using solution mappings we introduce our notion of *solutions* for a BGP:

[1] While we consider only BGPs in this paper, the solutions for BGPs that might be determined using link traversal based query execution, can be processed by the SPARQL algebra that provides operators for more complex SPARQL graph patterns.

[2] For the sake of a more straightforward formalization we do not permit blank nodes in BGPs as is possible according to the SPARQL specification [4]. In practice, each blank node in a SPARQL query can be replaced by a new variable.

Definition 2. *Let b be a BGP and let $\mathcal{D}$ be the set of all descriptor objects reachable by the execution of b. A solution mapping μ is a* solution *for b iff i) it holds*[3]*: $\mu[b] \subseteq \bigcup_{D \in \mathcal{D}} D$ and ii) μ maps only these variables that are in b, i.e. $\forall (v,t) \in \mu : v \in \bigcup_{tp \in b} vars(tp)$,*

2.3 Link Traversal based Construction of Solutions

While the two-phase definition approach in the previous section defines the notion of solutions for BGPs in the context of link traversal based query execution, it does not reflect the fundamental idea of intertwining link traversal with the construction of solutions. Instead, a query execution engine that would directly implement this two-phase approach would have to retrieve all reachable descriptor objects before it could generate solutions for a BGP. Hence, the first solutions could only be generated after all data links that correspond to triple patterns in the BGP have been followed recursively. Retrieving the complete set of reachable data can take a long time and may exceed the resources of the execution engine.

For this reason, the link traversal based query execution approach requires to construct the solutions incrementally, using a query-local dataset that is continuously augmented with additional descriptor objects. These descriptor objects are discovered by looking up URIs that occur in *intermediate solution*, that are, solution mappings from which the solutions are constructed. In the next section we discuss a possible implementation of this strategy.

3 Iterator Based Implementation

In [2] we introduce the idea of link traversal based query execution using an iterator based implementation of this idea. In this section we align this implementation approach with the general idea defined in the previous section. We give an introduction to the approach and discuss soundness and completeness.

3.1 Introduction to the Approach

Our implementation approach applies a synchronized pipeline of operators that evaluate a BGP in a fixed order. This pipeline is implemented as a chain of iterators $I_1, \ldots, I_n$ where each iterator I_i is responsible for triple pattern tp_i from the *ordered BGP*[4] $\bar{b} = [tp_1, \ldots, tp_n]$. The operation implemented by these iterators returns solution mappings that are solutions for a BGP consisting of the triple pattern of the corresponding iterator and all triple patterns of the

[3] For the union of descriptor objects we assume that no two descriptor objects share the same blank nodes. This requirement can be guaranteed by using a unique set of blank nodes identifiers for each descriptor object retrieved from the Web.

[4] We represent an ordered BGP as a list, denoted by comma-separated elements enclosed in brackets. In the remainder of this paper we conceive such an ordered BGP as a logical query plan. Selecting an order for a BGP is a query optimization problem as we discuss in Section 4. However, in this section we assume a given order.

Algorithm 1 `GetNext` function for our iterator based implementation approach.

Require: A set $\mathcal{D}$ of descriptor objects that always represents the current state of the query-local dataset; a triple pattern tp_i; a predecessor iterator I_{i-1} that provides solutions for $\{tp_1, \ldots, tp_{i-1}\}$ in $\mathcal{D}$; an initially empty set M_i that allows the iterator to keep matching triples between calls to this iterator function

1: **while** $M_i = \varnothing$ **do**
2: $\quad \mu' := I_{i-1}$.`GetNext`
3: $\quad$ **if** $\mu' =$ NotFound **then return** NotFound **end if**
4: $\quad tp'_i := \mu'[tp_i]$
5: $\quad$ **for all** $u \in uris(tp'_i)$ **do**
6: $\quad\quad$ **if** $lookup(u) \notin \mathcal{D}$ **then** $\mathcal{D} := \mathcal{D} \cup \{lookup(u)\}$ **end if**
7: $\quad$ **end for**
8: $\quad M_i :=$ set of all matching triples for tp'_i in $\bigcup_{D \in \mathcal{D}} D$
9: **end while**
10: $t_j :=$ an element in M_i
11: $M_i := M_i \setminus \{t_j\}$
12: $\mu_j :=$ a solution mapping such that $\mu_j[tp'_i] = t_j$ and $\forall (v,t) \in \mu_j : v \in vars(tp'_i)$
13: **return** $\mu' \cup \mu_j$

preceding iterators; i.e., each I_i provides solutions for $\{tp_1, \ldots, tp_i\}$. To determine these solutions each iterator executes the following three steps repetitively: First, the iterator consumes a solution mapping μ' of its direct predecessor[5] and applies this mapping to its triple pattern tp_i, resulting in a triple pattern $tp'_i = \mu'[tp_i]$ (lines 2 to 4 in Algorithm 1); second, the iterator ensures that the query-local dataset contains these descriptor objects that can be retrieved from looking up all URIs in tp'_i (lines 5 to 7); and, third, the iterator tries to generate solutions by finding matching triples for tp'_i in the query-local dataset (lines 8 to 13).

This approach is sound because each solution determined by the approach satisfies Definition 2 as we prove in [5]. Moreover, the approach is in fact an implementation of the idea of link traversal based query execution because it satisfies the criteria specified in Section 2.3: First, due to the second step, the iterators continuously augment the query-local dataset by looking up URIs on the Web and, second, all solutions are constructed incrementally, using this dataset.

The practicability of this approach is based on the following *look-up assumption*: If a solution mapping binds query variable v to URI u then the descriptor object $lookup(u)$ may contain matching triples for triple patterns that contain v. This assumption is justified by the common practice of publishing Linked Data.

3.2 Missing Query Results

Even if our iterator based approach is a correct implementation of link traversal based query execution, it does not guarantee to return all solutions that satisfy Definition 2. In the following we discuss the reasons for this limitation.

[5] We assume the first iterator, I_1, consumes a single, empty solution mapping once.

The most restricting characteristic of the iterator based approach is the fixed order in which it evaluates the triple patterns from a BGP. Since the discovery of reachable descriptor objects is aligned with the fixed-order evaluation of triple patterns, the approach cannot make use of the flexibility in the discovery as would be possible according to Definition 1. Hence, it may not discover all reachable descriptor objects and, thus, may miss some matching triples.

Additionally, the iterators dismiss an intermediate solution μ' consumed from their predecessor when they consume the next μ'. Hence, each μ' is used only once to find matching triples M_i for $\mu'[tp_i]$. Due to this "use and forget" strategy the approach misses matching triples for $\mu'[tp_i]$ that occur in these descriptor objects that will be discovered after the next μ' has been requested.

Finally, to enable an implementation that avoids inconsistencies and concurrency issues, the iterators determine all matching triples in isolation as represented by the set M_i in Algorithm 1. We propose to implement this strategy with an immutable snapshot of the query-local dataset [2,6]. However, by isolating the triple pattern matching, an iterator misses matching triples for $\mu'[tp_i]$ if they occur in descriptor objects that subsequent iterators discover when they consume the intermediate solutions generated from the current M_i, although the current μ' would still be available (in contrast to the aforementioned case).

Even if the iterator based approach may not return all solutions that satisfy Definition 2, it is worth studying: The effort to use this approach to enable link traversal based query execution in existing SPARQL query engines is comparably small, considering that the majority of engines use an iterator based execution strategy. Furthermore, its limitation may be accepted as a trade-off to avoid inapplicable long query execution times. Various Linked Data based applications employ the approach successfully; e.g., Researchers Map [7], Foaf Letter, AltMed[6], and an approach to consume distributed provenance traces [8].

3.3 The Impact of the Evaluation Order on Query Results

As a consequence of using a fixed evaluation order, the order which is actually being used influences which reachable descriptor objects a query engine discovers and, thus, which solutions it reports. The following example illustrates this effect:

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

Fig. 1. A sample BGP.

Example 1. To execute the BGP in Figure 1 we may select a query plan that uses the order given in the figure. During the execution of this plan the second iterator I_2 requests the first intermediate solution from its predecessor I_1. I_1 ensures that the query-local dataset contains the descriptor object $D_X = lookup($`http://.../X`$)$ [7] and, thereafter, I_1 tries to find matching triples for its triple pattern (i.e. the first pattern in Figure 1). Unfortunately, D_X does not contain such triples (cf. Figure 2). Hence, I_1 cannot provide an intermediate solution and, thus, the overall query result is empty. Even if we initialize the

[6] Find AltMed and Foaf Letter as part of http://www.linkeddata-a-thon.com

[7] For the sake of simplicity we assume the URIs at the predicate positions resolve to vocabulary definitions that do not contain relevant triples for our example.

Some of the data in $D_x = lookup(\texttt{http://.../X})$:	Some of the data in $D_b = lookup(\texttt{http://.../b})$:
<http://.../X> rdfs:subClassOf <http://.../Y> .	<http://.../b> rdfs:label "..." . <http://.../c> ex:p1 <http://.../b> .

Some of the data in $D_a = lookup(\texttt{http://.../a})$:	Some of the data in $D_c = lookup(\texttt{http://.../c})$:
<http://.../b> ex:p2 <http://.../a> .	<http://.../c> rdf:type <http://.../X> .

Fig. 2. Example descriptor objects and some of their data.

query-local dataset with all descriptor objects available from looking up the URIs in the query, i.e. D_X and $D_a = lookup(\texttt{http://.../a})$, we cannot find a matching triple for the first triple pattern. However, an alternative query plan could use the reverse order, i.e the first iterator is responsible for the last triple pattern in Figure 1. Executing this plan would result in *one* solution for the BGP: $\mu = \{(?\texttt{x}, \texttt{http://.../c}), (?\texttt{y}, \texttt{http://.../b}), (?\texttt{z}, \texttt{"..."})\}$.

As can be seen from the example, the iterator based approach may return different result sets for the same BGP depending on the evaluation order of the triple patterns in the BGP. This effect can be attributed to *missing backlinks* and *serendipitous discovery*, as we discuss in the following.

On the traditional, hypertext Web it is unusual that Web pages are linked bidirectionally. Similarly, an RDF triple of the form (uri_s, uri_p, uri_o) contained in $lookup(uri_s)$ (or $lookup(uri_o)$) does not have to be contained $lookup(uri_o)$ (or $lookup(uri_s)$). 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 1 is a missing backlink in D_X.

Following our look-up assumption each iterator retrieves descriptor objects because these objects may contain matching triples for the triple pattern tp'_i currently evaluated by the iterator. Thereby, all iterators augment the same query-local dataset. Thus, even if retrieved for the evaluation of a specific triple pattern such a descriptor object may also contain a triple t^* that matches another triple pattern tp'_j which will be evaluated later by any of the iterators. Since it is not guaranteed that the descriptor object with t^* is discovered and retrieved during the evaluation of tp'_j, we say that the solution generated based on t^* has been discovered by *serendipity*. If the BGP was ordered differently the descriptor object with t^* might only be discovered after tp'_j has already been evaluated and we could never generate the serendipitously discovered solution.

The effect of missing backlinks and serendipitous discovery on the number of query results is not a characteristic of link traversal based query execution in general; instead, it is specific to the iterator based implementation. In fact, this effect is a direct consequence of the restrictions discussed Section 3.2.

The dependency of result completeness on the order of a BGP implicates that certain orders are more suitable than others. Even if the iterator approach can not be guaranteed to return all solutions that satisfy Definition 2, selecting specific orders could provide for more solutions than other orders. In the next section we discuss the selection of execution orders.

4 Logical Query Planning

In this paper we understand an ordered BGP as a logical query plan. Basically, the creation of such a plan is the selection of a specific order for the triple patterns in a given BGP. Since there exist multiple orders it is possible to create different plans for the same BGP. In this section we discuss how to select one of these plans for the execution of the BGP: We consider the possibility to assess and rank query plans and argue that ranking-based plan selection is unsuitable in our scenario. As a consequence we propose a heuristic for plan selection.

4.1 Assessment of Query Plans

The query plans for a BGP have different characteristics, resulting in different execution performance. We assess them based on two criteria: cost and benefit.

The *cost* of a query plan can be measured in terms of, e.g., query execution time, the amount of network traffic caused, the number of URIs looked up, or the overall size of retrieved descriptor object. We use the query execution time to measure the cost because it provides for a more response time oriented plan assessment and it implicitly includes many of the other measures. For instance, the query execution time is dominated by delays resulting from the look-up of URIs, which may require a significant amount of time due to network latencies. While we propose approaches to reduce the impact of these delays [2], this impact can only be reduced but never be eliminated. Another factor that affects query execution time is the amount of retrieved data: With an increasing number of descriptor objects the time to find matching triples in the query-local dataset may increase, in particular, if each descriptor object is indexed separately [6].

Usually, traditional query optimization uses cost as the only selection criteria for query plans. However, in contrast to traditional query execution, the link traversal based execution of different plans for the same BGP may result in solution sets of different cardinality as we discuss in Section 3.3. Hence, for our iterator approach we should assess query plans not only based on their cost; instead, they must also be assessed by their *benefit*, that is, the number of solutions that an execution of the plan returns.

To rank and select query plans it is necessary to assess each of them without executing it. Since cost (and benefit) cannot be measured without execution, traditional query optimization techniques apply functions that calculate (or estimate) such measures. In our case it would be necessary to take the whole plan into account for such a calculation: The evaluation order of all triple patterns determines what intermediate solutions μ' an iterator I_i consumes from its direct predecessor, what triple patterns $\mu'[tp_i]$ it has to evaluate, what URIs it has to look up and, thus, which reachable descriptor objects it discovers. In this context it is important to note that even the construction of those intermediate solutions which cannot be used for the construction of solutions by subsequent iterators might be beneficial: These solution mappings might be necessary to discover descriptor objects that contribute to completely different solutions as the discussion of serendipitous discovery illustrates (cf. Section 3.3). Notice, due to these dependencies it is impossible to apply the popular dynamic programming approach [9] to generate optimal query plans.

We do not propose actual functions to calculate (or estimate) cost and benefit. Such a calculation requires information about reachable data and the data sources involved in the execution of a plan. In our scenario of link traversal based query execution we do not assume any of such information. We just have a query and an empty local dataset. Hence, before we start executing the query, we do not know anything about the descriptor objects we will discover; we do not even know what descriptor objects will be discovered. Based on this complete lack of information we could only assume a uniform distribution of input values for a cost (or benefit) function. The consequence would be an equal ranking of all possible query plans so that we could at best select a random plan. For this reason we propose to use a heuristic based approach that allows us to make at least an educated guess. 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. While we do not discuss such a strategy in this paper we will investigate adaptive query planning in the future.

4.2 Heuristic Based Plan Selection

Due to the complete lack of information at plan selection time the application of a cost (and benefit) based ranking of plans is unsuitable in our scenario. For this reason we propose to select query plans based on the following four rules:

- Dependency Respect Rule: Use a dependency respecting query plan.
- Seed TP Rule: Use a plan with a seed triple pattern.
- No Vocab Seed Rule: Avoid a seed triple pattern with vocabulary terms.
- Filtering TP Rule: Use a plan where all filtering triple patterns are as close to the seed triple pattern as possible.

These rules are based on our experience with the data that is currently available as Linked Data, on analyses of the queries executed with our prototypical query engine, and on our experience developing applications that use our query engine. In the remainder of this section we introduce and motivate these rules.

The Dependency Respect Rule proposes to use a *dependency respecting query plan*, that is, an ordered BGP in which at least one of the query variables in each triple pattern occurs in one of the preceding triple patterns. Formally, an ordered BGP $\overline{b} = [tp_1, \ldots, tp_n]$ is dependency respecting iff for each $i \in \{2, \ldots, n\}$ it holds: $\exists\, v \in vars(tp_i) : \big(\exists\, j < i : v \in vars(tp_j)\big)$. For BGPs which represents a connected graph[8] it is always possible to find a dependency respecting query plan. For the sake of simplicity, we assume all BGPs represent a connected graph.

Dependency respect is a reasonable requirement for query plans in our context because it enables each iterator to always reuse some of the bindings in intermediate solutions μ' 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.

[8] A BGP $b = \{tp_1, \ldots, tp_n\}$ represents a connected graph iff it holds:
$\forall b_1, b_2 \subset b : b_1 \cup b_2 = b \wedge b_1 \cap b_2 = \varnothing \wedge (\exists tp_i \in b_1, tp_j \in b_2 : vars(tp_i) \cap vars(tp_j) \neq \varnothing)$

The SEED TP RULE proposes to use a plan in which the first triple pattern is one of the *potential seed triple pattern*s, that are, triple patterns in the BGP which contain at least one HTTP URI. The rationale for using one of the potential seed triple patterns as the first pattern of a plan –which we then call the *seed triple pattern* of that plan– is the following: While query execution begins with an empty query-local dataset, any HTTP URI contained in the query may serve as a starting point to find matching triples. According to our look-up assumption (cf. Section 3.1), matching triples for a triple pattern might be found, in particular, in descriptor objects that were retrieved by looking up the URIs that are part of this pattern. Therefore, it is reasonable to select one of the potential seed triple patterns as the first triple pattern in the query plan.

The NO VOCAB SEED RULE proposes to avoid query plans with a seed triple pattern which contains only URIs that identify vocabulary terms. Such a URI can be identified with high likelihood by a simple syntactical analysis of a triple pattern: Since URIs in the predicate position are always vocabulary terms a preferred seed triple pattern must contain a URI in subject or object position. However, in triple patterns with a predicate of `rdf:type` a URI in the object position always identifies a class, i.e., also a vocabulary term. Hence, these triple patterns should also be avoided as seed triple patterns.

By narrowing down the set of query plans using the NO VOCAB SEED RULE we 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, according to our experience the majority of queries asks for instance data and does not contain patterns that have to match vocabulary definitions. Hence, it is reasonable to avoid seed triple patterns that are unlikely to link to instance data as a starting point for query execution. Example 1 illustrates the negative consequences of ignoring the NO VOCAB SEED RULE by selecting the `rdf:type` triple pattern as seed. Notice, for applications that mainly query for vocabulary definitions the rule must be adjusted.

The FILTERING TP RULE proposes to prefer query plans in which filtering triple patterns are placed as close to the seed 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 $\bar{b} = [tp_1, \ldots, tp_n]$ is a filtering triple pattern iff it holds: $\forall\, v \in vars(tp_i) : \big(\exists\, j < i : v \in vars(tp_j)\big)$.

The rationale of the FILTERING TP RULE is to reduce cost: During query execution, each intermediate solution μ' consumed by an iterator that is responsible for a filtering triple pattern tp_F, is guaranteed to contain bindings for all variables in tp_F. Therefore, the application of these μ' to tp_F 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'; otherwise, it discards this intermediate solution. Thus, the evaluation of filtering triple patterns may reduce the number of intermediate solutions but it will never multiply this number. Notice, for other triple patterns we cannot predict such a behavior, neither a reduction nor a multiplication of intermediate solutions.

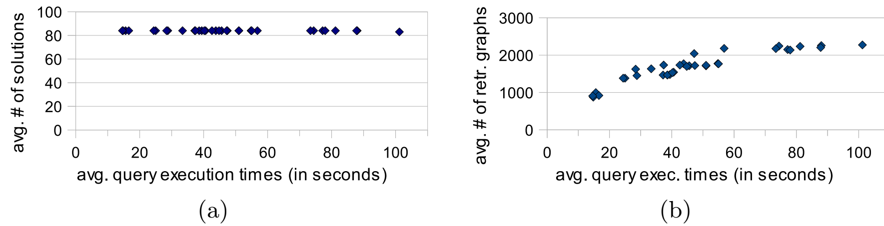

Fig. 3. Measurements for all query plans executed for the "Mylan" query in Figure 4.

The Filtering TP Rule is similar to the selection push-down that RDBMSs use to reduce the cost of query plans. However, for link traversal based query execution this rule might not always be beneficial because it reduces the likelihood for serendipitous discovery of matching triples and, thus, solutions (cf. Section 3.3). However, during the evaluation of our heuristic on the current Web of Data we did not experience such a hypothetical reduction of benefit.

5 Experimental Evaluation

In the previous section we argue that cost based plan selection is unsuitable for our iterator implementation of link traversal based query execution. As an alternative we propose a heuristic to select plans. In this section we evaluate the effectiveness of this heuristic.

5.1 Setup

For the evaluation we use three representative BGP queries. None of these queries can be answered using data from a single data provider alone. For each query we generated all dependency respecting query plans that have a seed triple pattern. We executed all these plans sequentially, using a new, initially empty query-local dataset for each plan. For each plan we measured the query exececution time, the number of retrieved descriptor objects, and the number of results. We ran each sequence of plans 6 times where the first run was for warm-up and was not considered for the measurements. These warm-up runs avoid measuring the effect of Web caches and enable servers that contributed data discovered during query execution to adapt their caches. The measurements of the other 5 runs were combined by calculating the arithmetic mean in order to minimize the potential for tampering the experiment by unexpected network traffic.

We executed the query plans using SQUIN[9] which is a prototype of a query engine that implements link traversal based query execution using the presented iterator approach. Retrieved descriptor objects are stored separately, each in a main memory index that contains six hashtables to support the typical access patterns[10] during triple pattern matching [6]. The experiment was conducted on

[9] http://squin.org

[10] subject given, predicate given, object given, subj.+pred., subj.+obj., pred.+obj.

an Intel Core 2 Duo T7200 processor with 2 GHz, 4 MB L2 cache, and 2 GB main memory. This machine was connected through the university LAN. Our test system runs a recent 32 bit version of Gentoo Linux with Sun Java 1.6.0.

5.2 Results

For the discussion of our measurements we mainly focus on the query in Figure 4. Nonetheless, we point out notable findings from the measurements taken for the other queries. The BGP in the query in Figure 4 contains 8 triple patterns; one of them qualifies as seed triple pattern according to the No Vocab Seed Rule. For this BGP exist 35 different, dependency respecting query plans with seed triple pattern, each of which has the `diseasome:name` triple pattern as filtering triple pattern.

```
SELECT ?cn ?bd2 WHERE {
 dailymed_orga:Mylan_Pharmaceuticals_Inc.
          dailymed:producesDrug ?bd .
 ?bd dailymed:genericDrug ?gd .
 ?gd drugbank:possibleDiseaseTarget ?dt .
 ?dt diseasome:name "Epilepsy" .
 ?bd dailymed:activeIngredient ?ai .
 ?bd2 dailymed:activeIngredient ?ai .
 ?c dailymed:producesDrug ?bd2 .
 ?c rdfs:label ?cn . }
```

Fig. 4. A SPARQL BGP query (prefix decl. omitted) that asks for companies which use the active ingredient of Mylan Pharmaceuticals' anti-epilepsy drug in their drugs as well.

Figure 3(a) illustrates the relationship between the average query execution times (QET) and the average number of solutions measured for the 35 plans. Each point in the chart represents a single plan. As can be seen from the chart, the number of discovered solutions, 84, was the same for all plans. However, the time required to determine these solutions differs significantly. To investigate the reasons that caused these differences we cluster the 35 plans by their QET into 4 groups. Table 1(a) summarizes statistics for these groups: the interval of QET that defines each group (*QET interval*), the number of plans in each group (*# of plans*), the arithmetic mean of the average number of descriptor objects retrieved by each plan in the group (*avg.#DO*), and the arithmetic mean of the position of the filtering triple pattern in each plan of the group (*avg.fTPpos*).

The *avg.fTPpos* values confirm the effectiveness of our heuristic, in particular, the Filtering TP Rule: The less efficient plans in groups G_3 and G_4 contain the filtering triple pattern in the seventh or eighth position; for the more efficient plans in group G_1 it is the fourth or fifth position, hence, closer to the seed triple pattern at the first position. The *avg.#DO* values indicate that during the execution of the less efficient plans more descriptor objects have been

Table 1. Statistics about groups of query plans for (a) the "Mylan" query in Figure 4 and (b) the "drug picture" query in Figure 5(a), grouped by the QET of the plans.

	G_1	G_2	G_3	G_4
QET interval	< 20s	[20s,35s]	[35s,60s]	> 60s
# of plans	4	5	18	8
avg.#DO	923.9	1494.0	1695.8	2208.5
avg.fTPpos	4.8	6.0	7.5	7.6

(a)

	G_1	G_2	G_3
QET interval	< 50s	[50s,100s]	> 100s
# of plans	4	3	3
avg.#DO	375.3	496.3	490.9
avg.fTPpos	4.8	6.0	6.0

(b)

retrieved than for the efficient plans. Figure 3(b) illustrates this observation in more detail by representing the plans individually. These measurements support the assumptions that motivated the proposal of the FILTERING TP RULE: The higher number of discovered descriptor objects indicates that more intermediate solutions have been processed by the less efficient plans.

For the other two queries we observed basically the same behavior. The second query, shown in Figure 5(a), contains 6 triple patterns; one qualifies as seed triple pattern according to the NO VOCAB SEED RULE. 10 dependency respecting query plans are possible, containing one filtering triple pattern each. The difference between the number of descriptor objects retrieved by the more and the less efficient plans is not as significant for this query as for the other queries (cf. the *avg.#DO* values in Table 1(b) and the corresponding chart in Figure 6). We attribute this to a low selectivity of the filtering triple pattern.

The third query, shown in Figure 5(b), contains 7 triple patterns of which two qualify as seed following the NO VOCAB SEED RULE. There are 56 dependency respecting query plans that can be grouped into two subsets, according to the selected seed triple pattern. Interestingly, all plans in one of these groups did not provide any solutions (cf. Figure 7). An investigation reveals that this problem can be attributed to a missing backlink; this backlink has not be discovered by starting the query execution with the seed triple pattern selected for all plans in the corresponding group. It was impossible to anticipate this problem automatically before executing the query. However, the plans that determined solutions ex-

Table 2. Statistics about the query plans for the "American Badger" query in Figure 5(b), grouped by the QET of the plans. The additional lines in this table list: the arithmetic mean of the average number of solutions determined by each plan in a group (*avg.#Sol*), the arithmetic mean of the triple patterns before the first filtering triple pattern in each plan of a group (*avg.b4fTP1*), the arithmetic mean of the triple patterns after the second filtering triple pattern in each plan of a group (*avg.afTP2*), and the arithmetic mean of the triple patterns between the two filtering triple patterns in each plan of a group (*avg.fTP1Δ2*).

	G_1	G_2	G_3	G_4
QET interval	<20s	[20s,70s]	[70s,130s]	>130s
avg.#Sol	0	0	28.1	27.6
# of plans	30	6	12	8
avg.#DO	13.0	15.6	205.1	309.3
avg.b4fTP1	3.03	3.33	3.67	4.50
avg.afTP2	0.63	0.00	0.67	0.25
avg.fTP1Δ2	1.33	1.67	0.67	0.25

```
SELECT ?gd2 ?p WHERE {
  ?gd db:drugCategory
       drugbank_category:antimalarials .
  ?gd db:brandedDrug ?bd .
  ?c dm:producesDrug ?bd .
  ?c rdfs:label "Pfizer Labs" .
  ?gd owl:sameAs ?gd2 .
  ?gd2 foaf:depiction ?p . }
```

(a)

```
SELECT ?s ?M WHERE {
  geospecies:4qyn7 gs:inFamily ?f .
  ?f skos:narrowerTransitive ?s .
  ?s skos:closeMatch ?m .
  ?m rdfs:subClassOf ?M .
  ?s gs:isExpectedIn ?loc .
  ?loc rdf:type gs:State .
  geospecies:4qyn7 gs:isExpectedIn ?loc .}
```

(b)

Fig. 5. Additional queries (prefix decl. omitted) used for the evaluation: (a) asks for pictures of generic drugs that are categorized as antimalarial and that are drugs, categorized as antimalarial and branded by "Pfizer Labs", (b) asks for species and their genus that are classified in the same family as the American Badger, *Taxidea taxus*, and that are expected in the same states as the American Badger.

hibit the expected behavior (cf. Table 2). As the only difference to the other two queries we note that each plan contains two filtering triple patterns.

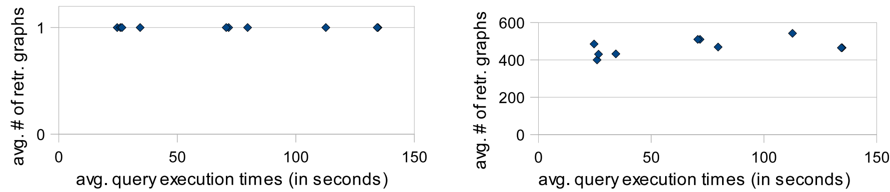

Fig. 6. Measurements for all query plans executed for the query in Figure 5(a).

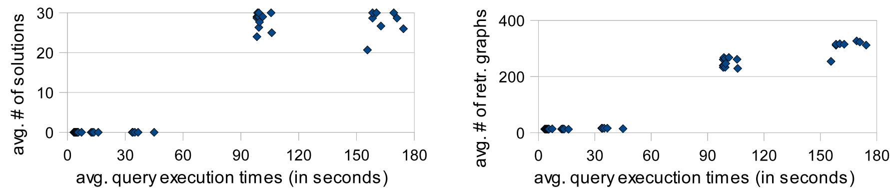

Fig. 7. Measurements for all query plans executed for the query in Figure 5(b).

6 Related work

In earlier work Mendelzon and Milo introduce an approach to execute SQL-like queries on the traditional, hypertext Web that includes the traversal of links [10]. They formalize the Web as a relational model and propose a two-phase approach to execute queries: First, all "reachable documents are retrieved, and then the query is evaluated on them." The same two-phase approach has been formalized by Bouquet et al. for the Web of Data [11]. While we also use two phases to define solutions in Section 2.2, the idea of link traversal based query execution is to intertwine query evaluation and link traversal instead of simply applying the two-phase approach for the actual execution of queries.

Harth et al. present an alternative approach that also uses URI look-ups to query the Web of data [12]. Instead of traversing links they use a data summary to identify descriptor objects that might be relevant for a query and should be retrieved for the execution. While this approach often performs better than our link traversal approach [3], it requires that all descriptor objects have been discovered, retrieved and summarized before queries can be executed. Furthermore, changes to the data of descriptor objects are not reflected in the summary.

There is only one other implementation approach for link traversal based query execution that we are aware of: In contrast to our synchronized pipeline of iterators, Ladwig and Tran recently proposed an asynchronous implementation

that uses symmetric hash joins [3]. While the authors report that their approach returns first results earlier, they measured the same overall query execution times for both approaches. We aim to analyze their approach in the context of the definitions presented here and compare it to our iterator based approach.

7 Conclusion

In this paper we analyze an iterator based implementation approach for the link traversal based query execution paradigm. We study its limitations and discuss how it benefits from a strategy that selects suitable query plans. Such a strategy must work without any statistics, data distribution records or other information about data and data sources because we do not assume any information when we start the execution of a query. Since traditional query planning techniques are unsuitable for this scenario we propose a heuristic for plan selection.

As future work we aim to develop our plan selection rules into a strategy that directly generates the most promising plans only. Furthermore, we investigate how to relax our zero knowledge assumption using information collected during previous query executions. Finally, the integration of adaptive query processing techniques shows great promise to improve our iterator based implementation.

References

1. Hartig, O., Langegger, A.: A database perspective on consuming Linked Data on the Web. Datenbank-Spektrum **10**(2) (2010)
2. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL queries over the web of linked data. In: Proc. of the 8th Int. Semantic Web Conference (ISWC). (2009)
3. Ladwig, G., Tran, D.T.: Linked data query processing strategies. In: Proc. of the 9th Int. Semantic Web Conference (ISWC2010). (2010)
4. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Rec., Online at http:// www.w3.org/TR/rdf-sparql-query/ (2008)
5. Hartig, O.: Iterator based implementations of link traversal based query execution. Online at http://squin.org/doc/IteratorImplementation/ (2010)
6. Hartig, O.: A main memory index structure to query linked data. In: Proc. of the 4th Int. Linked Data on the Web workshop (LDOW) at WWW. (2011)
7. Hartig, O., Mühleisen, H., Freytag, J.C.: Linked Data for building a map of researchers. In: Proc. of 5th Workshop on Scripting and Development for the Semantic Web (SFSW) at ESWC. (2009)
8. Hartig, O., Zhao, J.: Publishing and consuming provenance metadata on the Web of Linked Data. In: Proc. of the Int. Provenance and Annotation Workshop. (2010)
9. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proc. of the Int. Conference on Management of Data. (1979)
10. Mendelzon, A.O., Milo, T.: Formal models of Web queries. Information Systems **23**(8) (1998)
11. Bouquet, P., Ghidini, C., Serafini, L.: Querying the web of data: A formal approach. In: Proc. of the 4th Asian Semantic Web Conference (ASWC). (2009)
12. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K.U., Umbrich, J.: Data summaries for on-demand queries over Linked Data. In: Proc. of the 19th Int. Conference on World Wide Web (WWW). (2010)