How Caching Improves Efficiency and Result Completeness for Querying Linked Data

Olaf Hartig
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
hartig@informatik.hu-berlin.de

ABSTRACT

Link traversal based query execution is a novel query approach which enables applications that exploit the Web of Data to its full potential. This approach makes use of the characteristics of Linked Data: During query execution it traverses data links to discover data that may contribute to query results. Once retrieved from the Web, the data can be cached and reused for subsequent queries. We expect such a reuse to be beneficial for two reasons: First, it may improve query performance because it reduces the need to retrieve data multiple times; second, it may provide for additional query results, calculated based on cached data that would not be discoverable by a link traversal based execution alone. However, no systematic analysis exist that justifies the application of caching strategies based on these assumptions.

In this paper we evaluate the potential of caching to improve efficiency and result completeness in link traversal based query execution systems. We conceptually analyze the potential benefit of keeping and reusing retrieved data. Furthermore, we verify the theoretical impact of caching by conducting a comprehensive experiment that is based on a real-world application scenario.

1. INTRODUCTION

The emerging Web of Data [3] is a large dataspace that connects data from different providers who publish their data according to the Linked Data principles [2]. This dataspace opens possibilities not conceivable before: Applications are not restricted to data from a predefined set of sources; instead, new sources can be discovered and integrated at run-time. To enable applications to exploit the Web of Data to its full potential we propose *link traversal based query execution*, a novel approach to execute SPARQL queries over the Web of Data [12]. This approach makes use of the characteristics of Linked Data. The general idea is to intertwine query pattern matching with the traversal of data links in order to discover data that might be relevant to answer the executed query. We illustrate the application of this idea with an example:

EXAMPLE 1. The Linked Data based application *Foaf Letter*[1] allows users to observe and manage their FOAF[2] based social network. Foaf Letter uses a query engine that implements our link traversal based approach to issue SPARQL queries similar to Q1 (cf. Listing 1). Q1 asks for projects of acquaintances of user Bob, who is identified by URI http://bob.name. Link traversal based query execution typically starts with an empty, query-local dataset. We obtain some seed data for pattern matching by looking up URIs in the query: for the URI http://bob.name in Q1 we may retrieve a set G_b of RDF triples (cf. Figure 1), which we add to the local dataset. Now, we alternate between i) constructing intermediate solutions from RDF triples that match a pattern of our query in the query-local dataset and ii) augmenting the dataset by looking up URIs which are part of these intermediate solutions. For the triple pattern in line 2 of Q1 the local dataset contains a matching triple, originating from G_b. Hence, we can construct an intermediate solution $\mu_1 = \{$`?p` $\rightarrow$ http://alice.name$\}$ that maps query variable `?p` to the URI http://alice.name. By looking up this URI we may retrieve a set G_a of RDF triples, which we also add to the local dataset. Based on the augmented dataset we can construct an intermediate solution $\mu_2 = \{$`?p` $\rightarrow$ http://alice.name, `?pr` $\rightarrow$ http://.../AlicesPrj$\}$ for the pattern in line 3. Notice, constructing μ_2 is only possible because we discovered G_a based on μ_1. We proceed with our execution strategy: We retrieve G_p by looking up http://.../AlicesPrj and construct $\mu_3 = \{$`?pr` $\rightarrow$ http://.../AlicesPrj, `?l` $\rightarrow$ `"Alice's Project"`$\}$. Finally, we merge (i.e. join) μ_1, μ_2 and μ_3 to create a solution for the whole query.

```
1  SELECT ?p ?l WHERE {
2  <http://bob.name> foaf:knows ?p.
3  ?p foaf:currentProject ?pr .
4  ?pr rdfs:label ?l . }
```

Listing 1: Sample query Q1

Excerpt from RDF data G_b retrieved for `http://bob.name`:

```
<http://bob.name> foaf:knows <http://alice.name> .
```

Excerpt from RDF data G_a retrieved for `http://alice.name`:

```
<http://alice.name> foaf:name "Alice" ;
          foaf:knows <http://bob.name> ;
          foaf:currentProject <http://.../AlicesPrj> ;
          foaf:topic_interest <http://.../Tennis> .
```

Excerpt from RDF data G_p retrieved for `http://.../AlicesPrj`:

```
<http://.../AlicesPrj> rdfs:label "Alice's Project" .
```

Excerpt from RDF data G_t retrieved for `http://.../Tennis`:

```
<http://.../Tennis> rdfs:label "Tennis" .
```

Figure 1: Excerpts from Linked Data for the running example.

As the example demonstrates, link traversal based query execution proposes to evaluate a query over a dataset that is continuously augmented with potentially relevant data from the Web. The discovery of this data is driven by the URIs in intermediate solutions. Thus, the idea of the link traversal based approach is not to follow arbitrary links in the discovered data but only those links that correspond to triple patterns in the executed query.

[1]http://www.linkeddata-a-thon.com/index.php/FoafLetter
[2]http://www.foaf-project.org/

LDOW2011, March 29, 2011, Hyderabad, India.

Usually, not all the URIs looked up during query execution are in the same namespace, controlled by a single data provider. Instead, since the Linked Data principles require to provide links to data from other sources it is likely that data from multiple Linked Data-providing sources is discovered and used to construct query results; this may include sources the query engine did not even know they exist before executing the query. Hence, link traversal based query execution is fundamentally different from traditional query execution paradigms which always assume knowledge of the existence of data sources that may contribute to the query result. This assumption presents a restriction that inhibits applications to tap the full potential of the Web; link traversal based query execution, in contrast, enables serendipitous discovery and utilization of relevant data from unknown sources.

However, due to the openness and the widely distributed nature of the Web we cannot assume to find all data that is relevant for a query. Hence, we should never expect results that are complete w.r.t. the whole Web of Data. Nonetheless, users demand approaches that obtain as many results as possible. One possibility to improve completeness is to keep discovered and retrieved data and reuse it as additional seed data for subsequent queries as the following example illustrates:

EXAMPLE 2. Query Q2 (cf. Listing 2) asks for the interests of people who know Bob. The Foaf Letter application uses queries similar to Q2 to propose possible acquaintances for users. As for Q1 in Example 1, we start the execution of Q2 with an *empty* query-local dataset and we add G_b, discovered by looking up the URIs in Q2. Unfortunately, G_b does not contain triples that match the patterns in Q2 so that we cannot construct an intermediate solution. Hence, it is impossible to return a result for this query. However, instead of starting with an empty dataset, we could keep the data from the previous execution of Q1 and, thus, start executing Q2 with a query-local dataset that already contains G_b, G_a, and G_p. By using this data we can construct a query result with our link traversal based approach: $\mu_t = \{$`?name` $\rightarrow$ `"Alice"`, `?p` $\rightarrow$ http://alice.name, `?i` $\rightarrow$ http://.../Tennis, `?l` $\rightarrow$ `"Tennis"`$\}$.

```
SELECT ?name ?p ?i ?l WHERE {
 ?p foaf:knows <http://bob.name>.
 ?p foaf:name ?name.
 ?p foaf:topic_interest ?i .
 ?i rdfs:label ?l . }
```

Listing 2: Sample query Q2

As the example demonstrates, understanding the query-local dataset as a cache that can be reused for subsequent queries may benefit the result completeness for some of the queries. An additional argument for such a reuse is the typical purpose of caching: A cache may improve the efficiency of query executions. Such an expectation is based on the assumption that reusing formerly retrieved data may reduce the need for looking up URIs on the Web and, thus, decrease the amount of delays caused by these look-ups during query execution. While the advantage of using a cache for the link traversal based approach seems to be obvious, it requires a systematic investigation to understand how caching may improve result completeness and query performance. In particular, the first benefit that we expect, more complete result sets, must be researched because the discovery of a specific set of RDF triples is not guaranteed with our query approach, a characteristic which is untypical for most caching scenarios where all objects missing from the cache, can be obtained with some additional effort.

In this paper we evaluate the potential of caching for link traversal based query execution. This evaluation includes two parts: a conceptual analysis and an application based experiment. Hence, our main contributions are:

- We extend the original formalization of link traversal based query execution to provide a theoretical foundation to reuse the query-local dataset for the execution of multiple queries.
- Based on the extended formalization we conceptually analyze the impact of caching for link traversal based query execution.
- We empirically verify the theoretical impact of caching by conducting experiments that are based on a real world application scenario.

Notice, this paper does not focus on caching strategies such as replacement or invalidation. Instead, we understand the presented evaluation as a necessary first step to develop such strategies for our link traversal based approach.

The remainder of this paper is structured as follows. First, in Section 2 we introduce the preliminaries for the work presented in this paper, in particular, we present a formal definition of the idea of link traversal based query execution. Section 3 defines the integration of caching into our query approach and provides the conceptual analysis of the impact of caching. In Section 4 we present our experiment. Finally, Section 5 reviews related work and Section 6 concludes this paper.

2. DEFINITION OF LINK TRAVERSAL BASED QUERY EXECUTION

As a preliminary to a conceptual analysis of the potential benefit of caching for link traversal based query execution we formally introduce our query approach in this section. For the formalization we assume no changes are made to the data on the Web during the execution of a query.

Link traversal based query execution is a novel query approach to exploit the Web of data to its full potential. Since adhering to the Linked Data principles [2] is the minimal requirement for publication in the Web of Data, our query approach relies solely on these principles. Hence, we do not require each data publisher to provide a query interface (i.e. a SPARQL endpoint) for their dataset.

In the Web of Data entities have to be identified via HTTP scheme based URIs. Let U^{LD} be the set of all these URIs. By looking up such a URI we retrieve RDF data about the entity identified by the URI. Conceptually, we understand such a look-up as a function that refers to the resulting data: $lookup$ is a surjective function which returns for each URI $u \in U^{\mathrm{LD}}$ a *descriptor object*, that is, a set of RDF triples $\{(s_1, p_1, o_1), \dots (s_n, p_n, o_n)\}$ 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:

$$\forall u \in U^{\mathrm{LD}} : \Big(\exists (s,p,o) \in lookup(u) : s = u \vee o = u\Big)$$

Note, $lookup$ is not injective; the same descriptor object might be 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: $lookup(t) = \varnothing$.

EXAMPLE 3. The set G_b of RDF triples in Example 1 was retrieved by looking up the URI http://bob.name which identifies Bob. We understand G_b as a descriptor object that describes Bob. Hence, it holds: $lookup($ http://bob.name $) = G_b$.

We define link traversal based query execution for basic graph patterns[3] (BGPs), the basic building block of SPARQL queries [17]. A BGP is a set of triple patterns which are RDF triples that may contain query variables at the subject, predicate, and object position. For the sake of a more straightforward formalization we do not permit blank nodes in triple patterns as is possible according to the SPARQL specification. In practice, each blank node in a SPARQL query can be replaced by a new variable. A *matching triple* in a set of RDF triples G for a triple pattern $tp = (\tilde{s}, \tilde{p}, \tilde{o})$ is any RDF triple $(s, p, o) \in G$ with

$$(\tilde{s} \notin V \Rightarrow \tilde{s} = s) \wedge (\tilde{p} \notin V \Rightarrow \tilde{p} = p) \wedge (\tilde{o} \notin V \Rightarrow \tilde{o} = o)$$

where V denotes the set of query variables.

To formally integrate the traversal of data links into the query execution process we adjust the semantics of BGP queries. We define the notion of solutions for the link traversal based query execution of a BGP using a two-phase approach: First, we define the set of all descriptor objects that can be discovered during the link traversal based query execution of a BGP query. 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 (i.e. URI lookup) 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* by recursion:

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

- there exists a triple pattern $tp_i = (\tilde{s_i}, \tilde{p_i}, \tilde{o_i}) \in b$ such that $lookup(\tilde{s_i}) = D \vee lookup(\tilde{p_i}) = D \vee lookup(\tilde{o_i}) = D$; or
- there exists another descriptor object $D' \neq D$, a triple pattern $tp_j \in b$, 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_j in D', and iii) $lookup(s) = D$, $lookup(p) = D$, or $lookup(o) = D$.

EXAMPLE 4. The descriptor object G_b in Example 1 is reachable by the execution of the BGP in the sample query Q1 (cf. Listing 1) because G_b satisfies the first case in Definition 1: The first triple pattern in the BGP contains the URI http://bob.name and $G_b = lookup($ http://bob.name $)$. $G_a = lookup($ http://alice.name $)$ is also reachable; although, it satisfies the second case in Definition 1 because it depends on the reachability of G_b and on the matching triple therein. Similarly for G_p, which depends on G_a.

To represent the solutions of BGPs we adopt the notion of a *solution mapping* as defined in the SPARQL specification [17]. These mappings bind query variables to RDF terms. The application of a solution mapping μ to a BGP b, denoted as $\mu[b]$, implies replacing each variable in the triple patterns of b by the RDF term the variable is bound to in μ; unbound variables must not be replaced. Using solution mappings we introduce our notion of *solutions* for a BGP:

Definition 2. Let b be a BGP; let $\mathcal{A}$ be the set of all descriptor objects reachable by the execution of b. A solution mapping μ is a *solution* for b iff μ maps only these variables that are in b and it holds[4]:

$$\mu[b] \subseteq \bigcup_{D \in \mathcal{A}} D$$

While the presented two-phase definition approach 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 as we illustrate in Example 1. For a more detailed discussion of link traversal based query execution, including a comparison with other approaches that execute SPARQL queries over data from multiple sources on the Web, we refer to [13].

3. LINK TRAVERSAL BASED QUERY EXECUTION WITH A CACHE

The introductory Example 2 illustrates how link traversal based query execution may benefit from a *data cache* which keeps retrieved data (i.e. descriptor objects) and enables to reuse this data for subsequent queries. In this section we introduce a formalism to describe and analyze the impact of such a cache; based on this formalism, we provide an analysis of the potential benefit of caching. Notice, since we aim to analyze the potential of caching in general we assume the best case in this paper, that is, we assume an infinitely large memory to keep the query-local dataset so that no descriptor objects have to be replaced.

3.1 Formal Foundation

To conceptually enable the reuse of the query-local dataset for multiple queries we have to adjust the formalism introduced in the previous section. In particular, we have to extend our notion of reachability by introducing the integration of an already populated set of descriptor objects:

Definition 3. (Extension of Definition 1) Let $b = \{tp_1, \dots, tp_n\}$ be a BGP; let D be a descriptor object; let $\mathcal{D}_{\text{seed}}$ be a set of descriptor objects. D is *reachable* by the $\mathcal{D}_{\text{seed}}$*-initialized* execution of b iff either:

- $D \in \mathcal{D}_{\text{seed}}$;
- there exists a triple pattern $tp_i = (\tilde{s_i}, \tilde{p_i}, \tilde{o_i}) \in b$ such that $lookup(\tilde{s_i}) = D$, $lookup(\tilde{p_i}) = D$, or $lookup(\tilde{o_i}) = D$; or
- there exists another descriptor object $D' \neq D$, a triple pattern $tp_j \in b$, and an RDF triple $t = (s, p, o)$ such that i) D' is reachable by the $\mathcal{D}_{\text{seed}}$-initialized execution of b, ii) t is

[3]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 as usual.

[4]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.

a matching triple for tp_j in D', and iii) $lookup(s) = D$, $lookup(p) = D$, or $lookup(o) = D$.

This definition extends Definition 1 such that the reachability of a descriptor object depends on an initial content of the query-local dataset with which the query execution starts.

EXAMPLE 5. Example 2 illustrates the effect of the extended notion of reachability: Neither G_a nor G_t are reachable by the execution of the BGP in the sample query Q2, initialized with an empty set of descriptor objects. However, both, G_a and G_t, are indeed reachable by the $\mathcal{D}_{\text{Q1}}$-initialized execution of the BGP where $\mathcal{D}_{\text{Q1}} = \{G_b, G_a, G_p\}$ is the set of descriptor objects discovered and retrieved during the execution of Q1 as outlined in Example 1.

We note that our extended notion of reachability is monotone:

PROPOSITION 1. *Let b be an arbitrary BGP; let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two sets of descriptor objects where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. Each descriptor object that is reachable from the $\mathcal{D}_1$-initialized execution of b is also reachable from the $\mathcal{D}_2$-initialized execution of b.*

The proof of this proposition is trivial: It follows from the first case in Definition 3. From the proposition it follows also that each descriptor object that is reachable from the execution of a BGP b (according to Definition 1) is also reachable from any $\mathcal{D}$-initialized execution of b where $\mathcal{D}$ can be an arbitrary set of descriptor objects.

Based on our extended understanding of reachability we also extend our notion of solutions:

Definition 4. (Extension of Definition 2) Let b be a BGP; let $\mathcal{D}_{\text{seed}}$ be a set of descriptor objects; let $\mathcal{A}'$ be the set of all descriptor objects reachable by the $\mathcal{D}_{\text{seed}}$-initialized execution of b. A solution mapping μ is a $\mathcal{D}_{\text{seed}}$-*dependent solution* for b iff μ maps only these variables that are in b and it holds:

$$\mu[b] \subseteq \bigcup_{D \in \mathcal{A}'} D$$

EXAMPLE 6. The solution mapping μ_t discovered during the second query execution outlined in Example 2 was generated based on matching triples in G_a and G_t; i.e. $\mu_t[b_2] \subseteq G_a \cup G_t$ where b_2 is the BGP in Q2. Since G_a and G_t are reachable by the $\mathcal{D}_{\text{Q1}}$-initialized execution of b_2 (cf. Example 5) we note that μ_t is a $\mathcal{D}_{\text{Q1}}$-dependent solution for b_2.

The extended notion of solutions introduced in Definition 4 always includes all solution mappings that are solutions w.r.t. Definition 2, independent of the set of descriptor objects which a query execution was initialized with:

PROPOSITION 2. *Let b be an arbitrary BGP and $\mathcal{D}$ be an arbitrary set of descriptor objects. Each solution mapping μ that is a solution for b is also a $\mathcal{D}$-dependent solution for b.*

PROOF. Definition 2 assumes that each BGP is executed with an initially empty query-local dataset. Hence, each solution for a BGP is a $\mathcal{D}_{\text{empty}}$-dependent solution for that BGP where $\mathcal{D}_{\text{empty}} = \varnothing$ is the empty set of descriptor objects. Due to this fact, Proposition 2 follows from Proposition 1. □

As outlined before, the general idea of applying a data cache for the link traversal based execution of a sequence of queries is to reuse the data discovered during the execution of previous queries as seed data for subsequent queries. More precisely, we propose to reuse reachable descriptor objects from previous queries. Conceptually, we substitute the direct dependency of reachability on the initial content of the query-local dataset by an indirect dependency on the sequence of queries that have been executed before. Hence, we define which descriptor objects are reachable by the execution of a query sequence:

Definition 5. Let $B = [b_1, \dots, b_n]$ be a sequence of non-empty BGPs. The *set of reachable descriptor objects*, denoted as $\mathcal{R}(B)$, for the serial execution of B is recursively defined as:

$$\mathcal{R}(B) = \begin{cases} \varnothing & \text{; if } B = [\,] \text{ (i.e. if } B \text{ is empty)} \\ \mathcal{R}' & \text{; else} \end{cases}$$

where $\mathcal{R}'$ is the set of all descriptor objects that are reachable by the $\mathcal{R}(B')$-initialized execution of b_n, with $B' = [b_1, \dots, b_{n-1}]$.

EXAMPLE 7. Let $B_{\text{Q1}} = [b_{\text{Q1}}]$ be a BGP sequence that contains only the BGP b_{Q1} from our sample query Q1; let $B_{\text{Q1,Q2}} = [b_{\text{Q1}}, b_{\text{Q2}}]$ be a sequence that also contains the BGP b_{Q2} from query Q2. The set of reachable descriptor objects for these sequences are: $\mathcal{R}(B_{\text{Q1}}) = \{G_a, G_b, G_p\}$ and $\mathcal{R}(B_{\text{Q1,Q2}}) = \{G_a, G_b, G_p, G_t\}$. Furthermore, we can understand the μ_t discovered in Example 2 as a $\mathcal{R}(B_{\text{Q1}})$-dependent solution for b_{Q2}.

PROPOSITION 3. *Let B be a sequence of non-empty BGPs. For any arbitrary subsequence B' of B it holds: $\mathcal{R}(B') \subseteq \mathcal{R}(B)$.*

PROPOSITION 4. *Let b be an arbitrary BGP and B be an arbitrary sequence of non-empty BGPs. Each solution mapping μ that is a solution for b is also a $\mathcal{R}(B)$-dependent solution for b.*

Proposition 3 can be shown by induction using Definition 5 and Proposition 1; Proposition 4 follows from Definition 5 and Proposition 2. Please notice that Proposition 4 is particularly important because it shows that even if we reuse the query-local dataset for multiple queries it is always possible to determine at least these solutions for a BGP that satisfy our original Definition 2, independent of the query sequence executed beforehand.

The concepts defined in this section formally extend link traversal based query execution with the possibility to reuse the query-local dataset that contains descriptor objects retrieved during previous query executions. In particular, Definitions 4 and 5 provide the formal foundation to understand the query-local dataset as a data cache. In the following we make use of this formalization to conceptually analyze the potential benefit of caching.

3.2 The Potential Benefit of Caching

A data cache for descriptor objects serves two purposes in the context of link traversal based query execution: It may improve query performance and it may improve the completeness of results. In the remainder of this section we discuss these two purposes.

The first purpose of the data cache is the typical purpose of a cache: It may improve query performance by reducing the time required to execute queries. Query execution time depends on multiple factors such the number and size of descriptor objects that have to be retrieved, the time to actually retrieve these descriptor objects, and the time to find matching triples in the query-local dataset. To analyze the potential of caching we measure query performance by the number of descriptor objects that have to be retrieved from the Web because this number is the factor primarily affected by the use of a data cache.

Without caching the number of descriptor objects that have to be retrieved during the execution of a BGP b_{cur} is always[5] $\left|\mathcal{R}\left([b_{\text{cur}}]\right)\right|$.

[5] $[b_{\text{cur}}]$ denotes a sequence of BGPs that contains only b_{cur}

$$c\left(b_{\text{cur}},[b_1,\dots,b_n]\right) = \left|\mathcal{R}\big([b_{\text{cur}}]\big)\right| - \left|\mathcal{R}\big([b_{\text{cur}}]\big) \cap \mathcal{R}\big([b_1,\dots,b_n]\big)\right| + \left|\mathcal{R}\big([b_1,\dots,b_n,b_{\text{cur}}]\big) \setminus \Big(\mathcal{R}\big([b_{\text{cur}}]\big) \cup \mathcal{R}\big([b_1,\dots,b_n]\big)\Big)\right|$$

Figure 2: The *cost function* that calculates the number of descriptor objects which have to be retrieved during a link traversal based execution of BGP b_{cur} that reuses a local dataset already used for the execution of the BGP sequence $[b_1,\dots,b_n]$.

Using a data cache that was already used for the execution of a sequence $[b_1,\dots,b_n]$ of BGPs, may reduce this number if some of the $D \in \mathcal{R}([b_{\text{cur}}])$ have already been retrieved during the execution of any of b_1 to b_n. The exact reduction factor is $\left|\mathcal{R}([b_{\text{cur}}]) \cap \mathcal{R}([b_1,\dots,b_n])\right|$. The basic prerequisite to such a reduction is a temporal locality of descriptor objects: Descriptor objects must be reachable by the execution of multiple BGPs that are executed in a sequence. In this case a data cache eliminates the need to re-retrieve such descriptor objects and, thus, reduces query execution times. Hence, caching can indeed improve query performance. However, using a data cache may increase the number of reachable descriptor objects from $\left|\mathcal{R}([b_{\text{cur}}])\right|$ to $\left|\mathcal{R}([b_1,\dots,b_n,b_{\text{cur}}])\right|$. Hence, $\left|\mathcal{R}([b_1,\dots,b_n,b_{\text{cur}}]) \setminus \mathcal{R}([b_{\text{cur}}])\right|$ additional objects may become reachable. While some of them might also already be in the cache, others would have to be retrieved. The latter are exactly all $D \in \mathcal{R}([b_1,\dots,b_n,b_{\text{cur}}]) \setminus \big(\mathcal{R}([b_{\text{cur}}]) \cup \mathcal{R}([b_1,\dots,b_n])\big)$. Hence, the overall number of descriptor objects which have to be retrieved during the execution of b_{cur} using a data cache populated by the execution of $b_1,\dots,b_n$ can be calculated with the *cost function* in Figure 2.

EXAMPLE 8. For the second execution of Q2 in Example 2 we use our cost function to calculate $c(b_{\text{Q2}},[b_{\text{Q1}}])$ where b_{Q2} is the BGP of Q2 and b_{Q1} is the BGP of Q1 that was executed before.

$$\begin{aligned} c\left(b_{\text{Q2}},[b_{\text{Q1}}]\right) =& \left|\mathcal{R}\big([b_{\text{Q2}}]\big)\right| - \left|\mathcal{R}\big([b_{\text{Q2}}]\big) \cap \mathcal{R}\big([b_{\text{Q1}}]\big)\right| \\ &+ \left|\mathcal{R}\big([b_{\text{Q1}},b_{\text{Q2}}]\big) \setminus \Big(\mathcal{R}\big([b_{\text{Q2}}]\big) \cup \mathcal{R}\big([b_{\text{Q1}}]\big)\Big)\right| \\ =& \left|\{G_b\}\right| - \left|\{G_b\} \cap \{G_a,G_b,G_p\}\right| \\ &+ \left|\{G_a,G_b,G_p,G_t\} \setminus \Big(\{G_b\} \cup \{G_a,G_b,G_p\}\Big)\right| \\ =& 1 \end{aligned}$$

Hence, during the execution of Q2 we have to retrieve one additional descriptor object (G_t) if we reuse the local dataset with which we executed Q1.

We note that the application of a data cache improves query performance if $c(b_{\text{cur}},[b_1,\dots,b_n]) < \left|\mathcal{R}([b_{\text{cur}}])\right|$, that is, if the second factor in our cost function is greater than the third factor. Otherwise, i.e. if $c(b_{\text{cur}},[b_1,\dots,b_n]) > \left|\mathcal{R}([b_{\text{cur}}])\right|$, caching has a negative impact on query performance. While the latter case is disadvantageous w.r.t. query performance, it presents the price we have to pay to enable the second purpose of our data cache as we discuss in the following.

Example 2 illustrates that the application of a data cache may improve result completeness. To measure such an improvement we introduce a completeness criterion:

Definition 6. Let b be a BGP; let Ω_{all} be the set of all $\mathcal{D}_{\text{all}}$-dependent solutions for b, where $\mathcal{D}_{\text{all}}$ is the (potentially infinite) set of all descriptor objects that can be retrieved from the Web. Furthermore, let $\mathcal{D}$ be an arbitrary set of descriptor objects and Ω the set of all $\mathcal{D}$-dependent solutions for b. Ω is *complete* w.r.t. all data on the Web iff $\Omega = \Omega_{\text{all}}$. The *degree of completeness* of Ω is:

$$completeness(\Omega) = \begin{cases} 1 & ;\text{ if } |\Omega_{\text{all}}| = 0 \\ \frac{|\Omega|}{|\Omega_{\text{all}}|} & ;\text{ else} \end{cases}$$

Hence, we understand a set of solutions for a BGP as complete if it contains all solutions that could be constructed if we had all descriptor objects that can be retrieved from the Web. However, in practice it is impossible to decide whether a set of solutions is complete because we cannot discover all descriptor objects. Hence, it is also impossible to calculate the degree of completeness. Nonetheless, we can compare different sets of solutions for a BGP w.r.t. their degree of completeness by comparing their cardinality. For a BGP b_{cur} executed with a query-local dataset that was already used for the execution of the BGP sequence B, we calculate this cardinality by the following *benefit function*:

$$b\left(b_{\text{cur}},B\right) = \left|\big\{\mu \mid \mu \text{ is a } \mathcal{R}(B)\text{-dependent solution for } b_{\text{cur}}\big\}\right|$$

Since any set of solutions for a BGP b_{cur} includes the solutions that can be determined without caching (cf. Proposition 4) it always holds $b(b_{\text{cur}},B) \geq b(b_{\text{cur}},[\,])$, independent of B. The remaining difference

$$s\left(b_{\text{cur}},B\right) = b\left(b_{\text{cur}},B\right) - b\left(b_{\text{cur}},[\,]\right)$$

corresponds to these solutions that can only be determined using at least one of the descriptor objects in $\mathcal{R}([b_1,\dots,b_n,b_{\text{cur}}]) \setminus \mathcal{R}([b_{\text{cur}}])$. Since these descriptor objects only become reachable with a data cache and, hence, we discover the corresponding $s(b_{\text{cur}},B)$ solutions by serendipity, we call $s(b_{\text{cur}},B)$ *serendipity factor*. For query executions for which the serendipity factor is greater than 0 the application of a data cache improves result completeness.

Table 1 summarizes the possible effect of caching on link traversal based query execution. In this table we use the serendipity factor as an indicator for the impact on result completeness and a *performance factor*

$$p\left(b_{\text{cur}},B\right) = c\left(b_{\text{cur}},[\,]\right) - c\left(b_{\text{cur}},B\right)$$

as an indicator for the impact query performance.

Table 1: Indicators for the impact of caching on query performance and result completeness as can be observed for the link traversal based execution of BGP b_{cur} that reuses a query-local dataset which was used for the serial execution of the BGP sequence B before.

Criterion	Indicator	Impact
Query Performance	$p(b_{\text{cur}},B) < 1$	negative
	$p(b_{\text{cur}},B) = 0$	none
	$p(b_{\text{cur}},B) > 1$	improvement
Result Completeness	$s(b_{\text{cur}},B) = 0$	none
	$s(b_{\text{cur}},B) > 0$	improvement

EXAMPLE 9. For the first execution of Q2 in Example 2 we calculate the following impact factors:

$$\begin{aligned} p\left(b_{\text{Q2}},[\,]\right) &= c\left(b_{\text{Q2}},[\,]\right) - c\left(b_{\text{Q2}},[\,]\right) = 1 - 1 = 0 \\ s\left(b_{\text{Q2}},[\,]\right) &= b\left(b_{\text{Q2}},[\,]\right) - b\left(b_{\text{Q2}},[\,]\right) = 0 - 0 = 0 \end{aligned}$$

For the second execution, for which we reuse the query-local dataset with which we executed Q1 before, we calculate:

$$p\left(b_{\mathrm{Q2}},[b_{\mathrm{Q1}}]\right)=c\left(b_{\mathrm{Q2}},[\,]\right)-c\left(b_{\mathrm{Q2}},[b_{\mathrm{Q1}}]\right)=1-1=0$$
$$s\left(b_{\mathrm{Q2}},[b_{\mathrm{Q1}}]\right)=b\left(b_{\mathrm{Q2}},[b_{\mathrm{Q1}}]\right)-b\left(b_{\mathrm{Q2}},[\,]\right)=1-0=1$$

4. EXPERIMENTAL EVALUATION OF THE BENEFIT OF CACHING

To verify the theoretical benefit of caching as discussed in the previous section, we conducted comprehensive experiments that are based on a real world application scenario. In this section we present these experiments: We describe the setup and the experiments and discuss the measured values.

4.1 Setup

To evaluate the impact of caching in a realistic scenario it is necessary to use the query workload of a real world application. For our experiments we selected the Foaf Letter application (cf. Example 1) which offers three main functionalities: (F1) Presentation of a user's acquaintances and their main properties; (F2) Proposal of additional types of properties that the user may want to specify; and (F3) Proposal of other acquaintances not listed in a user's FOAF file.

To provide these functionalities Foaf Letter issues the following five types of SPARQL queries:

Contact Queries of this type ask for various properties of a user's acquaintances.

UnsetProps Queries of this type ask for properties that are specified for acquaintances of a user, but not for the user; restricted to properties from the FOAF vocabulary.

Incoming Queries of this type ask for "incoming contacts" of a user; i.e. persons who know the user that are not listed to be known by the user.

2ndDegree1 Queries of this type ask for persons who are known by at least two distinct acquaintances of a user, but who are not listed to be known by the user.

2ndDegree2 Queries of this type ask for persons who are "incoming contacts" for at least two distinct acquaintances of a user, but who are not listed to be known by the user.

We selected Foaf Letter for the experiments because it is an example of an application that largely benefits from our link traversal based query execution approach. Answers to queries of the listed types require data from multiple providers, published at different locations on the Web. For instance, Contact queries can only be answered using data describing acquaintanceships and data representing the properties of acquaintances. While the former is typically provided in a user's FOAF file, the latter can usually be obtained from the FOAF files of the acquaintances. Notice, since it is unknown who will use Foaf Letter and what their acquaintances are, the application of traditional query approaches is unsuitable for Foaf Letter because they assume a fixed set of potentially relevant data sources is given in advance. Our link traversal based approach, in contrast, provides the necessary flexibility to discover relevant data on the fly.

For our experiments we defined actual SPARQL queries of the five types for 23 persons who publish their FOAF data according to the Linked Data principles. Using the resulting 115 queries we specified a *query sequence* that simulates a scenario in which all 23 persons use the Foaf Letter application, one after another; each use includes functionalities F1, F2, and F3, in this order. Hence, the query sequence orders all 115 queries by the corresponding person for whom they are defined. Each of the five queries for each person are ordered by their type: Contact, UnsetProps, 2ndDegree1, 2ndDegree2 and Incoming. While we used this, fixed query sequence for the majority of experiments we also simulated a less predictable workload for which we randomly re-ordered the sequence.

During the experiments we executed the queries using the Semantic Web Client Library[6] (SWClLib). SWClLib is a prototype of a query engine that implements link traversal based query execution. Retrieved descriptor objects are stored separately, each in a main memory index that contains six hashtables to support the typical types of triple patterns[7] during triple pattern matching. To simulate a data cache in our experiments we simply reuse the query-local dataset represented by a set of such indexes instead of re-populating this set for each query.

For each query execution we measured the number of results returned, the overall query execution time, the number of descriptor objects retrieved during the query execution, the number of descriptor objects that the data cache contained after query execution, and the hit rate. We determined the *hit rate* using the formula

$$\text{hit rate}=\frac{\text{number of hits}}{\text{number of look-up requests}}$$

where *number of look-up requests* is the number of all URI look-ups requested during query execution and *number of hits* is the number of those URI look-up requests that could be answered from the cache (i.e. where the corresponding descriptor object $lookup(u)$ has already been retrieved during a previous query execution). Notice, due to the implementation of our query approach in SWClLib some URI look-ups are requested more than once during a query execution so that even with an empty cache we measured hit rates greater than 0.

The experiments were conducted on an Intel Core 2 Duo T7200 processor with 2 GHz, 4 MB L2 cache, and 2 GB main memory. This machine is connected through the university LAN and runs a recent 32 bit version of Gentoo Linux with Sun Java 1.6.0. The actual SPARQL queries as well as the software used to run our experiments, the measurements, and all the diagrams created from these measurements can be found on the Website[8] for this paper.

4.2 Experiments

For our analysis we conducted four experiments: lower bound, upper bound, given order, and random orders.

During the *lower bound* experiment we executed each query of our query sequence with a new, initially empty query-local dataset. Hence, we did not reuse the dataset for multiple queries. With this experiment we obtained measurements for applying our link traversal based approach *without* caching. These measurements represent a baseline for our analysis. The experiment includes three runs of the query sequence; for our analysis we use the average of the values measured for each query.

For the *upper bound* experiment we executed each query of the query sequence three times, reusing the query-local dataset for these three executions, and we took the measurements for the third execution, only. For each query, however, we used a new, initially empty, query-local dataset. Hence, this experiment measured the impact of caching for single queries.

[6]http://www4.wiwiss.fu-berlin.de/bizer/ng4j/semwebclient/
[7]subj. given (s), pred. given (p), obj. given (o), s+p, s+o, p+o
[8]http://squin.org/experiments/CachingLDOW2011/

Table 2: Average number of results reported for the queries in the four experiments (standard deviation in parentheses).

experiment	all queries	Contact queries only	UnsetProps queries only	2ndDegree1 queries only	2ndDegree2 queries only	Incoming queries only
lower bound	4.983 (11.658)	18.478 (20.937)	3.478 (3.301)	3.130 (6.137)	0.391 (0.941)	0.000 (0.000)
upper bound	5.070 (11.813)	18.435 (20.876)	3.522 (3.287)	3.130 (6.137)	0.391 (0.941)	0.000 (0.000)
given order	6.878 (12.158)	19.130 (21.192)	4.348 (4.217)	3.130 (6.137)	5.130 (6.811)	2.652 (3.845)
random orders	6.652 (11.966)	18.739 (20.946)	3.870 (3.721)	3.435 (6.316)	4.609 (6.287)	2.435 (3.894)

Table 3: Average hit rate measured during the execution of the queries in the four experiments (std. dev. in parentheses).

experiment	all queries	Contact queries only	UnsetProps queries only	2ndDegree1 queries only	2ndDegree2 queries only	Incoming queries only
lower bound	0.576 (0.182)	0.500 (0.086)	0.734 (0.128)	0.647 (0.172)	0.610 (0.175)	0.387 (0.105)
upper bound	0.996 (0.017)	0.994 (0.010)	0.984 (0.033)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)
given order	0.932 (0.139)	0.784 (0.220)	0.977 (0.795)	0.980 (0.069)	0.963 (0.075)	0.954 (0.077)
random orders	0.954 (0.036)	0.945 (0.025)	0.974 (0.019)	0.958 (0.030)	0.948 (0.034)	0.946 (0.054)

The *given order* experiment executes the whole query sequence with the same, query-local dataset. Hence, this experiment allows to evaluate the potential benefit of caching for the execution of multiple queries, in particular, for later queries in the sequence. We ran the query sequence three times, each sequence with a new, initially empty, query-local dataset, and we use the average of the measurements for each query in the analysis.

The specified query sequence represents a sequential and rather predictable use of the Foaf Letter application by multiple users. Other applications may show other access patterns that result in less well ordered query sequences. We simulate such a behavior with the *random orders* experiment. This experiment corresponds to 30 runs of the given order experiment with different query sequences. For each of these 30 runs we use a random permutation of the original sequence. The measurements from these runs for each query are averaged for the analysis.

Before we executed the four experiments we performed a warm-up, that is, a single run of the lower bound experiment. This warm-up is necessary for two reasons: 1.) to avoid measuring the effect of Web caches and 2.) to enable Linked Data servers that contributed descriptor objects discovered during query execution to adapt their caches.

4.3 Discussion of the Measurements

In the remainder of this section we discuss the measurements obtained from our experiments. The charts in Figure 4 put the values measured for all queries during the different experiments into relation. The order of the queries in these charts, from top to bottom, corresponds to the order in which we executed these queries. Tables 2 to 4 provide a summary of the measurements. These tables show the average number of query results (cf. Table 2), the hit rate (cf. Table 3), and the average query execution times (cf. Table 4) for all four experiments, aggregated over all queries and over the different types of queries.

4.3.1 lower bound *vs.* upper bound

Figure 4(a) depicts the number of query results reported for each query (summarized in Table 2). For a few queries we measured a slightly different number of query results in the lower bound and the upper bound experiment as the small difference in the average number of results in Table 2 indicates. The reasons for these small deviations are measurement errors caused by network timeouts: Some URI look-ups timed out during some of the executions of the corresponding queries. Due to these timeouts the query system sometimes missed a few reachable descriptor objects and, thus, was not able to construct some of the query results. The average hit rates in Table 3 confirm this explanation: In the ideal case we would expect a hit rate of 1 for the upper bound experiment but the average hit rate is slightly below 1 which indicates that even after the second re-execution of some of the queries the query-local dataset did not contain all reachable descriptor objects. However, if we leave out these measurement errors we see that the number of query results is equal in the lower bound and the upper bound experiments. We attribute this finding to the fact that even if we reuse the query-local dataset for multiple executions of the same query we do not reach more descriptor objects and, thus, cannot construct more solutions. This effect can also be seen in Figure 4(c) which illustrates that the number of descriptor objects contained in the query-local dataset after each query execution is also equal for the two experiments.

While caching cannot enable more complete result sets in this case, it may benefit query performance as can be seen in Table 4 and Figure 4(b): For several queries the execution times were significantly higher in the lower bound experiment than in the upper bound experiment; for 47 of the 115 queries the difference was larger than 30 seconds. This effect can be explained by comparing the measured hit rates (cf. Figure 4(d), summarized in Table 3). While the hit rates during the upper bound experiment were almost[9] always 1, the hit rates in the lower bound experiment were significantly smaller. This observation supports our expectation that the higher number of URI look-ups needed in the lower bound experiment (which corresponds to a smaller performance factor) increased the amount of delays caused by these look-ups and, thus, had a negative impact on query execution times. Hence, for multiple executions of a single query a data cache always improves query performance but it has no impact on result completeness. On the level of BGPs we explain this finding formally: Since it holds

$$\mathcal{R}([b_{\text{cur}}]) = \mathcal{R}([b_{\text{cur}}, b_{\text{cur}}])$$

for each BGP b_{cur}, we derive $c(b_{\text{cur}}, [b_{\text{cur}}]) = 0$ and thus:

$$p(b_{\text{cur}}, [b_{\text{cur}}]) = c(b_{\text{cur}}, [\,])$$

Furthermore, we also derive $b(b_{\text{cur}}, [b_{\text{cur}}]) = b(b_{\text{cur}}, [\,])$ and thus:

$$s(b_{\text{cur}}, [b_{\text{cur}}]) = 0$$

[9]The few outliers correspond to the measurement errors discussed before.

Table 4: Average query execution time measured for the queries in the four experiments (standard deviation in parentheses).

experiment	all queries	Contact queries only	UnsetProps queries only	2ndDegree1 queries only	2ndDegree2 queries only	Incoming queries only
lower bound	30.036 (46.708)	52.919 (83.019)	30.465 (22.404)	43.585 (43.393)	18.947 (19.847)	4.228 (8.342)
upper bound	1.943 (11.375)	0.016 (0.026)	8.317 (24.568)	1.331 (3.518)	0.049 (0.131)	< 0.000 (< 0.000)
given order	39.845 (145.898)	38.904 (88.739)	109.510 (295.164)	31.850 (82.635)	18.181 (35.337)	0.780 (1.024)
random orders	36.994 (118.7)	7.257 (9.194)	117.115 (240.865)	38.310 (69.939)	19.573 (27.814)	2.715 (4.927)

4.3.2 given order

While caching cannot increase the number of results for single queries it can indeed improve result completeness for a sequence of queries as the measurements from the given order experiment indicate (cf. Figure 4(a) and Table 2). For 53 of the 115 queries the query engine reported more results in the given order experiment than in the upper bound or lower bound experiment. These 53 queries are of the type Incoming or 2ndDegree2. For queries of these types query answering requires descriptor objects that are not reachable from the execution over an initially empty query-local dataset. For instance, each Incoming query requires data (in particular foaf:knows statements) about persons who are not mentioned in the data about the current user. While the data about the current user can be discovered and retrieved, starting from the URI of the user, the required data about these other persons can not be discovered because there is no link to them in the data about the user. If, however, data about these other persons has already been retrieved during the execution of a previous query (maybe because these persons used the Foaf Letter application themself or they were also relevant for Contact or UnsetProps queries performed for other users) the query engine can use this data to return results for the current Incoming query. For the other query types the cache does not increase result completeness (ignoring the aforemention measurement errors) because all descriptor objects necessary to answer these queries are also reachable with an initially empty query-local dataset.

While understanding the query-local dataset as a cache that can be reused for multiple queries may benefit result completeness it also has an impact on query performance. Figure 4(b) (summarized in Table 4) illustrates that the query execution time for all queries was higher in the given order experiment than in the upper bound experiment. This is not surprising because all relevant data was already in cache in the upper bound experiment while for the given order experiment several URI look-ups were necessary as the hit rates indicate (cf. Figure 4(d) and Table 3).

Comparing the given order experiment to the lower bound experiment instead, we see that caching benefitted query performance in several cases: 26 of the 115 queries had a significantly higher (more than 30 seconds) execution time without the cache. However, more interesting are 12 queries that had a significantly (more than 30 seconds) higher execution time in the given order experiment than in the lower bound experiment. Our investigation revealed that for some of these queries the time to find matching triples in the query-local dataset dominates the time required for URI look-ups. Hence, a query-local dataset that contains many descriptor objects which are irrelevant for these queries (as was the case in the given order experiment) causes a significant overhead. Primarily, this effect can be attributed to the representation of the query-local dataset in the current implementation of our query engine. Storing each descriptor object in a separate index requires multiple index look-ups during triple pattern matching, one look-up for every descriptor object in the query-local dataset. To avoid this issue we are working on a new data structure that allows to keep data from multiple descriptor objects in a single index. However, the other reason for a higher execution time of some of these 12 queries are cases in which the performance factor is negative.

4.3.3 random orders

The measurements from the random orders experiment indicate that caching can also benefit the less predictable workload of the randomly permuted query sequences. On the average, the random orders experiment showed a similar behavior as the given order experiment. However, for some of the queries the random orders experiment reveals result completeness improvements that could not be measured in the given order experiment. Figure 3 illustrates this behavior exemplarily for all queries of type Incoming. In the randomly ordered query sequences, these queries are more often executed later than in the our original query sequence and, thus, can take more advantage of the data cache than in the given order experiment.

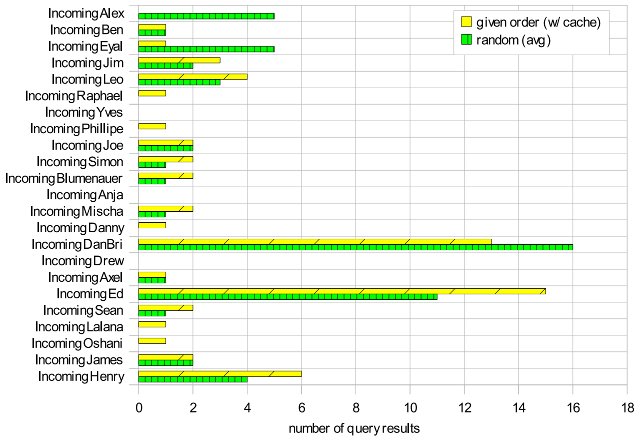
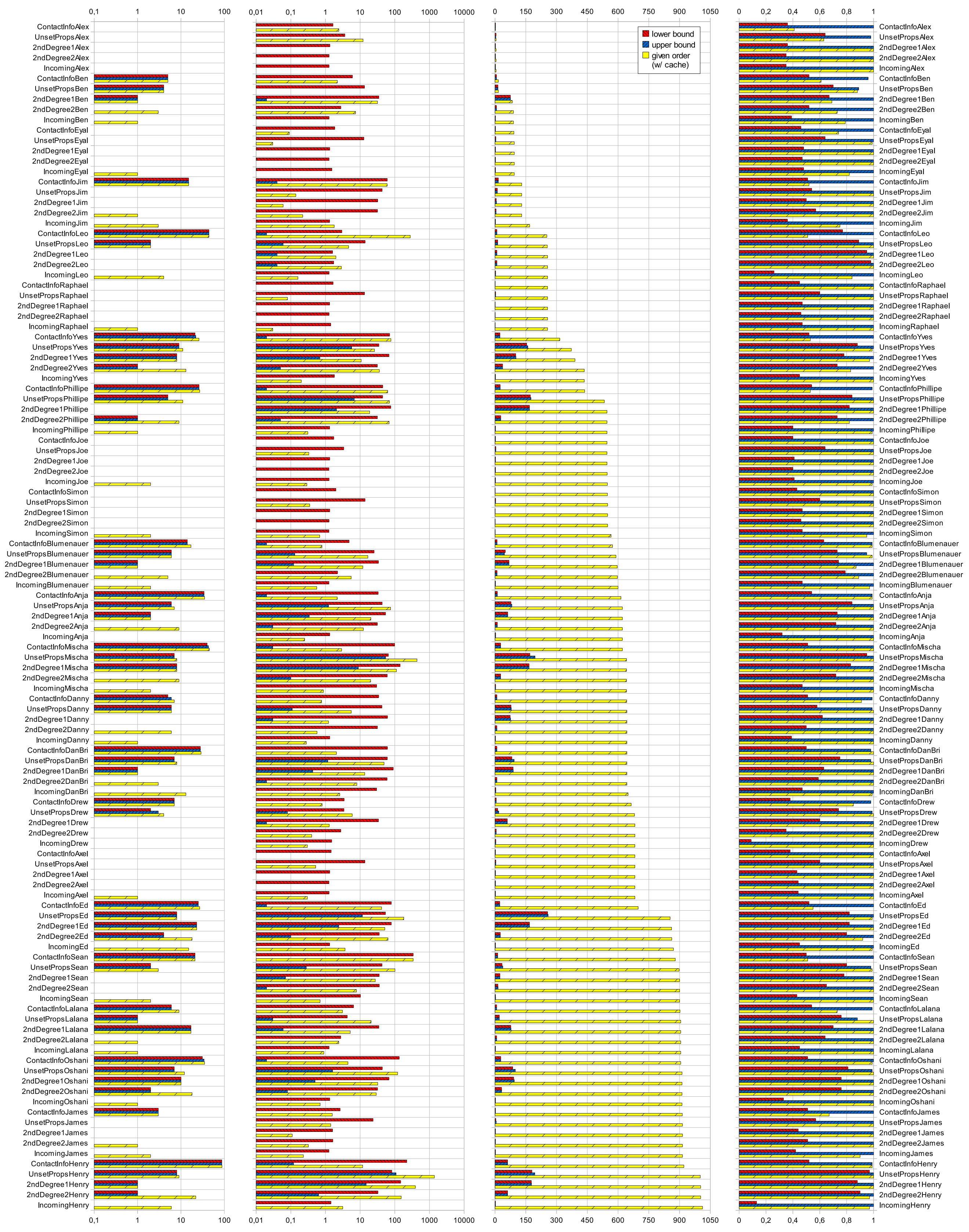


Figure 3: Comparison of the number of query results reported for queries of type Incoming during the given order and the random orders experiment.

5. RELATED WORK

To the best of our knowledge no work exists that addresses the topic of caching in the context of link traversal based query execution. What comes closest is the work presented by Harth et al. [11]. The authors introduce an alternative approach that also uses URI look-ups to query the Web of Data. Instead of traversing links they use a data summary to identify descriptor objects that might be relevant for a query and, thus, should be retrieved for the execution. While this approach often performs better than our link traversal approach [16], it requires that all descriptor objects have been discovered, retrieved and summarized before queries can be executed. This requirement inhibits serendipitous discovery and utilization of initially unknown data sources as is possible with our approach.



(a) number of query results (b) query exec. time in seconds (c) descriptor objects in cache (d) hit rate

Figure 4: Main measurements for all queries in the lower bound, upper bound, **and** given order **experiments.**

However, a data summary can be understood as an alternative to a data cache in our context: Each descriptor object that is discovered (and used) during the link traversal based execution of a query can additionally be summarized. The data summary that would emerge from such a strategy could be used to obtain additional seed data for the execution of subsequent queries. Moreover, a hybrid approach that combines a data cache with a data summary would be possible: When a descriptor object has to be replaced from the cache because additional space is required, the object could be summarized and not just be deleted. We aim to investigate these ideas in the future.

Web caching in general is an extensively studied topic because it minimizes access latency, reduces network traffic, and causes a reduced workload on Web servers [1, 8, 19]. Information systems that consume Web resources also benefit from caching because it allows for an immediate availability of resources [5, 18]. Furthermore, there is a lot of work that studies client side data caching to improve the performance of applications in general (e.g. [14]) and of query processing in particular (e.g. [6, 9]). However, the only advantage of caching in these studies is an increased efficiency to complete certain tasks. In our scenario, in contrast, data caching may also improve the quality of the results of such tasks, that is, result completeness of query executions in our case.

While this paper focuses on data caching, it is also possible to cache query results (or parts thereof). This idea has been studied primarily in the context of client-server database systems and distributed database environments (e.g. [4, 7, 10, 15]). Similar to the aforementioned data caching approaches for query processing, the purpose of caching query results in such systems is to improve query efficiency. We expect a similar effect from adding a query result cache to a link traversal based query system.

6. CONCLUSION

In this paper we analyze the potential of caching for link traversal based query execution, a novel query approach which exploits the Web of Linked Data to its full potential. Our analysis shows an advantage that is untypical for traditional applications of caching: In our scenario a data cache may provide for additional query results. Furthermore, our analysis verifies the possible benefit of a data cache for improving query performance as one would expect from the application of caching. However, we also identify cases in which caching has a negative impact on the query performance. We understand this drawback as the trade-off for which we gain the possibility to discover more query results.

We note, however, that a beneficial impact of caching on result completeness and query performance as measured in our experiments, is only possible in certain cases. In general, it requires semantically similar queries that access data about the same topic, describing the same entities. More precisely, these queries have to benefit from data that is reachable from queries executed previously. The possibility to increase the number of query results is only given if the reachable data from the executed query sequence contains required data that is not reachable for the current query with an initially empty query-local dataset.

While this paper presents a first, systematic evaluation that verifies the benefit of adding a data cache to a link traversal based query system, we ignore the issues of replacement and invalidation for our analysis. However, the presented formalization provides the theoretical foundation to develop and to analyze replacement and invalidation strategies. In the future we will work on concepts to actually integrate a data cache in our query system.

7. REFERENCES

[1] G. Barish and K. Obraczka. World Wide Web caching: Trends and Techniques. *IEEE Communications Magazine*, 38, 2000.

[2] T. Berners-Lee. Linked Data. Online at http://www.w3.org/DesignIssues/LinkedData.html, 2006.

[3] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data – The Story So Far. *Journal on Semantic Web and Information Systems*, 5(3), 2009.

[4] C.-M. Chen and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT)*, 1994.

[5] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[6] S. Cluet, O. Kapitskaia, and D. Srivastava. Using LDAP Directory Caches. In *Proceedings of the 18th Symposium on Principles of Database Systems (PODS)*, 1999.

[7] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, 1996.

[8] B. D. Davison. A Web Caching Primer. *IEEE Internet Computing*, 5(4), 2001.

[9] M. J. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[10] P. Godfrey and J. Gryz. Answering Queries by Semantic Caches. In *Proceedings of the 10th Int. Conference on Database and Expert Systems Applications (DEXA)*, 1999.

[11] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data Summaries for On-Demand Queries over Linked Data. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010.

[12] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *Proceedings of the 8th International Semantic Web Conference (ISWC)*, 2009.

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

[14] A. Iyengar. Design and Performance of a General-Purpose Software Cache. In *Proceedings of the International Performance Computing and Communications Conference (IPCCC)*, 1999.

[15] A. M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal*, 5(1), 1996.

[16] G. Ladwig and D. T. Tran. Linked Data Query Processing Strategies. In *Proceedings of the 9th International Semantic Web Conference (ISWC)*, 2010.

[17] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, Online at http://www.w3.org/TR/rdf-sparql-query/, 2008.

[18] P. Triantafillou, N. Ntarmos, and J. Yannakopoulos. A Cache Engine for E-Content Integration. *IEEE Internet Computing*, 8(2), 2004.

[19] J. Wang. A Survey of Web Caching Schemes for the Internet. *ACM Computer Communication Review*, 29(5), 1999.