A Main Memory Index Structure to Query Linked Data

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

Frank Huber
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
huber@informatik.hu-berlin.de

ABSTRACT

A possible approach to query Linked Data combines the actual evaluation of a query with the traversal of data links in order to discover and retrieve potentially relevant data. An implementation of this idea requires approaches that support an efficient and flexible management of temporary, ad hoc data collections that emerge during query execution. However, existing proposals for managing RDF data primarily focus on persistent storage and query execution for large datasets and, thus, are unsuitable in our dynamic scenario which involves many small sets of data.

In this paper we investigate main memory data structures to store Linked Data in a query execution system. We discuss the requirements for such a data structure, introduce three strategies that make use of hash table based indexes, and compare these strategies empirically. While this paper focuses on our query execution approach, the discussed data structures can also be applied to other approaches that retrieve Linked Data from remote sources via URI look-ups in order to process this data locally.

1. INTRODUCTION

In [12] we propose *link traversal based query execution* as a new query execution paradigm tailored for the Web of Data. In contrast to traditional query execution paradigms which assume knowledge of a fixed set of potentially relevant data sources beforehand, our approach conceives the Web of Data as an initially unknown set of data sources. We make use of the characteristics of Linked Data, in particular, the existence of links between data items of 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 allows the execution engine to discover potentially relevant data during the query execution.

An important characteristic that distinguishes link traversal based query execution from other approaches, such as query federation, is the retrieval and query-local processing of Linked Data from the Web. Moreover, the integration of result construction and link traversal means that the query-local data collection is continuously augmented with further data discovered during the query execution. To support this procedure a link traversal based query system requires a data structure that stores the retrieved data while it is being processed. Such a data structure must allow the query operators to efficiently add and access the retrieved data. For our query execution approach, such an access includes the evaluation of triple based queries. Furthermore, the data structure must also support concurrent access in order to enable an efficient, multi-threaded implementation of our query approach. Disk-based data structures for RDF[1] are unsuitable for the task because they feature very costly I/O operations and there is no need to persist data collections that emerge during link traversal based query execution. However, existing data structures that manage RDF data in main memory do not satisfy the outlined requirements as well, primarily because they are optimized for path queries [14, 22] or for complete graph queries [2, 5, 16] but not for simple triple based queries.

In this paper we investigate three main memory data structures that satisfy the outlined requirements. All three data structures are based on the same index structure for RDF. This index uses hash tables which store sets of RDF triples, arranged in six possible ways to efficiently support all possible types of triple based queries. The three studied data structures, which we call *individually indexed representation*, *combined index representation*, and *combined quad index representation*, differ in how they actually make use of the index structure: The individually indexed representation stores the data resulting from each link traversal step in a separate index, whereas, the combined and the quad index representation use a single index for all data that the query engine retrieves during the execution of queries. These different, index utilization strategies result in different characteristics of the three data structures: A single index enables a more efficient execution of triple based queries than a collection of individual indexes. On the other hand, it is not as straightforward to add, remove, or replace data in a single index that multiple, parallel link traversal operations access concurrently than it is for the individually indexed representation. However, the combined index representation and the quad index representation that we propose in this paper support concurrent access in an efficient manner. Although we present and discuss the three data structures in the context of link traversal based query execution, they can also be applied to other approaches that retrieve Linked Data by looking up URIs in order to process this data locally; for instance, the navigational query approach proposed by Bouquet et al. [7] or the data summary based query approach from Harth et al. [10]. Hence, our main contributions are:

- three main memory data structures for ad hoc storing of Linked Data that is consecutively retrieved from remote sources in order to be processed locally, including a discussion how to access these data structures, and
- an evaluation that analyzes the proposed data structures and compares them empirically.

This paper is structured as follows: First, we introduce preliminaries in Section 2, including the link traversal based query execution paradigm. In Section 3 we discuss the requirements that a data structure must support in order to be suitable for our usage

[1]RDF is the data model employed by Linked Data.

scenario. Based on these requirements we review existing work in Section 4. Section 5 introduces the studied data structures that satisfy our requirements, including the index structure they are based on. We empirical evaluate these data structures in Section 6 and conclude in Section 7.

2. PRELIMINARIES

This section introduces the preliminaries for the work presented in this paper, including a brief, informal introduction of our query approach: Link traversal based query execution is a novel query execution paradigm developed to exploit the Web of Data to its full potential. Since adhering to the Linked Data principles [3] is the minimal requirement for publication in the Web of Data, link traversal based query execution relies solely on these principles instead of assuming the existence of source-specific query services.

The Linked Data principles require to describe data using the RDF data model. RDF distinguishes three distinct sets of *RDF terms*: URIs, literals, and blank nodes. Let U be the 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 $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$.

In the Web of Data each entity has to be identified via a single HTTP scheme based URI. 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. Conceptually, we understand the result of such a URI look-up as a descriptor object:

Definition 1. A *descriptor object* is a set of RDF triples, i.e. a finite subset $D \subset (U \cup B) \times U \times (U \cup B \cup L)$, which i) was retrieved by looking up the URI $u \in U^{\mathrm{LD}}$ of an arbitrary entity and which ii) describes that entity. To denote the URL of the Web resource from which a descriptor object D was actually retrieved we write $url(D)$.

The query language for RDF data is SPARQL [17] which is based on graph patterns and subgraph matching. The basic building block of a SPARQL query is a *basic graph pattern (BGP)*, that is, a finite subset of the set $(U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$ where V is an infinite set of query variables, distinct from U, B and L. The elements of a BGP are called *triple pattern*s. We adjusted the semantics of BGP queries[2] for our link traversal based query execution approach. While we outline the idea in the following, we refer to [11] for a formal definition.

To provide results for a BGP query, a link traversal based query execution engine intertwines the construction of such query results with the traversal of data links in order to discover data that might be relevant to answer the executed query. By using the descriptor objects retrieved from looking up the URIs in a query as a starting point, a link traversal based query execution engine evaluates a certain triple pattern of the BGP. The intermediate solutions resulting from this triple pattern evaluation usually contain further URIs. These URIs link to additional data which may provide further, intermediate solutions for the same or for other patterns of the BGP. To determine results of the whole BGP query the execution engine alternates between evaluating triple patterns and looking up URIs; finally, the intermediate solutions for the triple patterns have to be joined. A comprehensive example of link traversal based query execution can be found in [13], in which we also compare our approach to other approaches that execute SPARQL queries over the Web of Data.

[2]While we consider only BGP queries in this paper, the results for a BGP that might be determined using link traversal based query execution, can be processed by the SPARQL algebra which provides operators for the other types of graph patterns in SPARQL queries.

As can be seen from the outlined procedure, instead of evaluating a BGP query over a fixed set of RDF triples, our link traversal based approach executes BGPs over a dataset that is continuously augmented with descriptor objects retrieved during query execution itself. Formally, we define such a query-local dataset as follows:

Definition 2. A *query-local dataset* is a set of descriptor objects where for each two distinct descriptor objects D_1 and D_2 in the query-local dataset it must hold $url(D_1) \neq url(D_2)$.

3. REQUIREMENTS

In this section we discuss the requirements for a data structure that stores the query-local dataset. We first describe the core functional requirements, introduce important non-functional properties, and, finally, mention properties that are not necessary.

3.1 Core Functional Requirements

The data structure must store a query-local dataset and it must support the following four operations over the query-local dataset:

- find – This operation finds RDF triples in the query-local dataset that match a given triple pattern. Formally, an RDF triple $t = (s, p, o)$ is a *matching triple* for a triple pattern $(\tilde{s}, \tilde{p}, \tilde{o})$ in a query-local dataset $\mathcal{D}$ iff it holds[3] $t \in \bigcup_{D \in \mathcal{D}} D$ and

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

  Due to this definition the find operation must report each matching triple only once, even if it occurs in multiple descriptor objects.

- add – This operation adds a given descriptor object to the query-local dataset. The query execution system performs this operation for each descriptor object that has been retrieved during the execution of a query.

- remove – The remove operation removes a descriptor object that was retrieved from a given URL from the query-local dataset. This operation is necessary to enable an invalidation strategy in query systems that (re-)use the query-local dataset for the execution of multiple queries. While such a reuse can be beneficial w.r.t. query execution costs and result completeness [11], it bears the risk of providing query results based on stale data. Thus, an invalidation strategy should identify potentially stale descriptor objects and remove them from the query-local dataset.

- replace – This operation replaces a specific descriptor object with a more recent version that has been retrieved from the same URL. This operation is useful for invalidation strategies that immediately retrieve new versions of (potentially) stale data.

Due to the dynamic nature of link traversal based query execution it is likely that an implementation of this query approach makes use of multiple processes (or threads). For instance, our prototypical query system implements URI look-ups by asynchronous function calls and, thus, enables the processing of multiple look-up tasks in parallel [12]. Due to this parallelism multiple processes may access the query-local dataset concurrently. Therefore, 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.

data structure for storing the query-local dataset has to enable the implementation of the aforementioned operations in a way that it supports concurrent access. In particular, we require the transactional property isolation, given that an execution of any of the four operations, find, add, remove and replace, can be understood as a transaction over the stored query-local dataset. Isolation requires that "events within a transaction must be hidden from other transactions running concurrently" [8]. In our case, this requirement means that the addition, replacement and removal of descriptor objects must not have side effects that cause an interference with any other add, remove and replace operation. Furthermore, each find operation must operate on a single, immutable state of the query-local dataset, where:

- *only* those RDF triples that are part of a descriptor object contained in the query-local dataset can be used for triple pattern matching (i.e. by the find operation), and
- *all* RDF triples that are part of a descriptor object contained in the query-local dataset can be used for triple pattern matching.

The isolation requirement also implies that:

- a descriptor object can only be used for triple pattern matching after it has been added completely,
- different descriptor objects which were retrieved from the same URL must not be used at the same time for triple pattern matching, and
- a descriptor object cannot be used for triple pattern matching when it has already been removed partially.

3.2 Non-Functional Requirements

The data structure that stores a query-local dataset has to support an efficient execution of queries. The query execution time of link traversal based query execution depends on multiple factors such as delays caused by looking up URIs, retrieving descriptor objects and adding them to local data structures, as well as the time to actually evaluate the query patterns over the query-local dataset. In order to reduce the latter two factors we require a data structure that supports an efficient implementation of the operations find, add and replace. In particular, the performance of the find operation should scale with the number of descriptor objects in the query-local dataset. Furthermore, the overall amount of main memory consumed by the data structure storing a query-local dataset should be as small as possible to allow for query-local datasets that contain a large number of descriptor objects.

3.3 Non-Relevant Properties

In addition to the aforementioned requirements a data structure that can hold Linked Data from the Web may have further properties and other approaches to consume Linked Data may have different requirements. In the remainder of this section we point out potentially relevant properties that are not required in our scenario.

In our case it is not necessary to query specific descriptor objects individually: Intermediate solutions in link traversal based query execution are generated from matching RDF triples that may occur in any of the descriptor objects. For this reason, we understand the find operation to operate on the whole query-local dataset instead of individual descriptor objects. Therefore, it is also not necessary that an implementation of find distinguishes and reports the descriptor objects from which matching triples originate. However, the latter may become a requirement for systems that have to provide the provenance of each query result.

Since the execution of queries is a read-only process there is no need to write data back to the Web. It is not even necessary to modify retrieved descriptor objects within the query system.

Finally, our understanding of a transaction in this paper does not correspond to the traditional understanding according to which a whole query execution is a single transaction. In the case of link traversal based query execution we do not require transactional properties on that level. In fact, multiple queries executed in parallel may even mutually benefit from a shared and collectively augmented query-local dataset because descriptor objects discovered and retrieved for one query could also be useful to others.

4. RELATED WORK

Several data structures to store RDF data have been proposed in the literature. In this section we review them w.r.t. the requirements introduced in the previous section.

The majority of existing work focuses on *RDF stores*, which are DBMSs that are designed to persistently store large amounts of RDF data in secondary memory. Theoharis et al. provide a classification and comparison of earlier approaches that make use of relational databases [21]. More recent work proposes to partition the data vertically by grouping RDF triples with the same property into dedicated tables and to store these tables in a column-oriented DBMS [1], an approach that is suitable for data with a moderate number of properties [19]. An alternative to relational storage schemes is the development of native, disk-based data structures for RDF data (e.g. [20, 15, 23]). From the work in this area the most relevant in our context is the index structure presented by Harth and Decker [9]. The authors associate RDF triples (s, p, o) with a context c and represent such a "*triple in context*" by a *quad*, that is, a 4-tuple (s, p, o, c). The proposed index structure comprises a lexicon and 6 quad indexes. The lexicon provides a mapping from RDF terms to identifiers in order to save space and improve processing time. The quad indexes are implemented using B+ trees and contain quads encoded by the identifiers of their elements. While all 6 quad indexes contain all indexed quads, each index uses different elements of the quads as index keys such that these 6 quad indexes cover all kinds of quad based queries, leveraging the fact that B+ trees support prefix queries. While this data structure is disk-based, its design influenced the main memory data structures that we investigate in this paper.

In contrast to RDF stores, the majority of existing software frameworks for RDF, such as Jena[4] and Sesame[5], also provide an in-memory data structure for RDF data. In our experiments we empirically compare some of these in-memory models with the data structures discussed in this paper (cf. Section 6). However, to the best of our knowledge, only a few publications exist that explicitly introduce data structures to manage RDF data in main memory: Atre et al. introduce BitMat [2], a compressed bit-matrix structure that enables processing of BGP queries on very large sets of RDF triples completely in main memory. Binna et al. pursue the same goal with a main memory RDF store, called SpiderStore [5]. SpiderStore applies a graph-based storage layout that represents an RDF graph (i.e. a graph presentation of a set of RDF triples) natively. Based on this representation, Binna et al. propose a depth-first graph traversal approach to evaluate BGP queries. Since the applied graph-based storage layout is optimized for graph traversal operations, query execution in SpiderStore is very efficient. However, due to the focus on storing a single RDF graph and executing complete BGP queries, both approaches, BitMat and SpiderStore,

[4]http://openjena.org/
[5]http://openrdf.org/

are unsuitable in our context; we require a data structure that is optimized for an efficient evaluation of single triple patterns over multiple RDF graphs.

Oren et al. propose the application of evolutionary algorithms to execute queries over RDF data [16]. The authors represent the data in main memory using Bloom filters. While this approach allows the query system to process large amounts of data, the proposed query execution method also focuses on an evaluation of complete BGPs instead of triple pattern queries. Furthermore, the approach provides for approximate query answers only.

Janik and Kochut present BRAHMS, a main memory-based storage system for RDF [14]. BRAHMS is based on a read-only data structure which comprises multiple hash tables, similar to the index structure that we use. However, BRAHMS requires these hash tables to enable an efficient retrieval of node neighborhoods in RDF graphs in order to support path queries for semantic association discovery algorithms. Another approach that focuses on path queries over RDF graphs has been presented by Udrea et al. [22] who propose a graph based index called GRIN. While both approaches, BRAHMS and GRIN, provide for an efficient query execution, they are unsuitable in our context, due to their focus on path queries.

5. STORING QUERY-LOCAL DATASETS

In this paper we investigate three main memory data structures that satisfy the requirements as introduced in Section 3. All three data structures are based on the same index for RDF data. This index enables an efficient execution of triple pattern queries (i.e. find operations). In this section we introduce the index and describe the three data structures.

5.1 Indexing RDF Data

The index that we introduce in the following, stores a set of RDF triples; it comprises a dictionary, a triple list, and 6 hash tables.

The *dictionary* provides a two-way mapping between RDF terms and numerical identifiers for these terms. Using term identifiers allows for a more efficient query execution because it is faster to process and compare numbers than strings or any other kind of object representation. Formally, we represent the dictionary by two bijective functions: $id : (U \cup B \cup L) \rightarrow I$ and $term : I \rightarrow (U \cup B \cup L)$ where I denotes the set of numerical term identifiers; id and $term$ are inverse to each other. Based on the term identifiers we introduce the notion of an *ID-encoded triple*, that is a 3-tuple containing three identifiers of RDF terms. To denote the ID-encoded triple that corresponds to an RDF triple $t = (s, p, o)$ we write $enc(t)$; i.e. $enc(t) = (id(s), id(p), id(o))$; similarly, $dec(\bar{t})$ denotes the RDF triple corresponding to an ID-encoded triple $\bar{t} = (\bar{s}, \bar{p}, \bar{o})$; i.e. $dec(\bar{t}) = (term(\bar{s}), term(\bar{p}), term(\bar{o}))$.

In addition to the dictionary, the index contains a *triple list* and 6 *hash tables*, denoted as H_S, H_P, H_O, H_SP, H_SO and H_PO. Each of these hash tables consists of n different hash buckets. To denote the i-th bucket in hash table H_X we write $H_\mathrm{X}[i]$. These buckets store references to ID-encoded triples; all triples referenced in the buckets are stored in the *triple list* of the index. The hash function $h_\mathrm{X} : I \times I \times I \rightarrow [1, n]$ of each of the 6 hash tables associates any ID-encoded triple $\bar{t}$ with the bucket which may contain a reference to $\bar{t}$ in that hash table.

The index contains 6 hash tables to support the different access patterns based on which the query execution system may try to find RDF triples that match a triple pattern. These access patterns correspond to the 8 different kinds of triple patterns (cf. Table 1 in which '?' denotes that the corresponding element of the triple pattern is a query variable and '!' denotes the element is an RDF term). To support these access patterns each of the 6 hash tables has to contain

Table 1: Relevant hash buckets for triple patterns (s, p, o) of different types.

Pattern Type	Relevant Bucket(s)
$(!, ?, ?)$	$H_\mathrm{S}[h_\mathrm{S}(id(s), id(p), id(o))]$
$(?, !, ?)$	$H_\mathrm{P}[h_\mathrm{P}(id(s), id(p), id(o))]$
$(?, ?, !)$	$H_\mathrm{O}[h_\mathrm{O}(id(s), id(p), id(o))]$
$(!, !, ?)$	$H_\mathrm{SP}[h_\mathrm{SP}(id(s), id(p), id(o))]$
$(!, ?, !)$	$H_\mathrm{SO}[h_\mathrm{SO}(id(s), id(p), id(o))]$
$(?, !, !)$	$H_\mathrm{PO}[h_\mathrm{PO}(id(s), id(p), id(o))]$
$(!, !, !)$	e.g. $H_\mathrm{SO}[h_\mathrm{SO}(id(s), id(p), id(o))]$
$(?, ?, ?)$	e.g. $\bigcup_{i=1}^{n} H_\mathrm{S}[i]$

references to all ID-encoded triples of the RDF data to be indexed; furthermore, the corresponding hash functions, h_S, h_P, h_O, h_SP, h_SO and h_PO, have to satisfy the following requirement: For each pair of ID-encoded triples $(\bar{s_1}, \bar{p_1}, \bar{o_1})$ and $(\bar{s_2}, \bar{p_2}, \bar{o_2})$ it holds:

$$\begin{aligned}
\bar{s_1} = \bar{s_2} &\Rightarrow h_\mathrm{S}(\bar{s_1}, \bar{p_1}, \bar{o_1}) = h_\mathrm{S}(\bar{s_2}, \bar{p_2}, \bar{o_2}) \\
\bar{p_1} = \bar{p_2} &\Rightarrow h_\mathrm{P}(\bar{s_1}, \bar{p_1}, \bar{o_1}) = h_\mathrm{P}(\bar{s_2}, \bar{p_2}, \bar{o_2}) \\
\bar{o_1} = \bar{o_2} &\Rightarrow h_\mathrm{O}(\bar{s_1}, \bar{p_1}, \bar{o_1}) = h_\mathrm{O}(\bar{s_2}, \bar{p_2}, \bar{o_2}) \\
\bar{s_1} = \bar{s_2} \wedge \bar{p_1} = \bar{p_2} &\Rightarrow h_\mathrm{SP}(\bar{s_1}, \bar{p_1}, \bar{o_1}) = h_\mathrm{SP}(\bar{s_2}, \bar{p_2}, \bar{o_2}) \\
\bar{s_1} = \bar{s_2} \wedge \bar{o_1} = \bar{o_2} &\Rightarrow h_\mathrm{SO}(\bar{s_1}, \bar{p_1}, \bar{o_1}) = h_\mathrm{SO}(\bar{s_2}, \bar{p_2}, \bar{o_2}) \\
\bar{p_1} = \bar{p_2} \wedge \bar{o_1} = \bar{o_2} &\Rightarrow h_\mathrm{PO}(\bar{s_1}, \bar{p_1}, \bar{o_1}) = h_\mathrm{PO}(\bar{s_2}, \bar{p_2}, \bar{o_2})
\end{aligned}$$

Due to this requirement we can expect to find all triples that may match a triple pattern (s, p, o) of type $(!, ?, ?)$ from searching through the references in the $h_\mathrm{S}(id(s), id(p), id(o))$-th bucket of hash table H_S; i.e. in bucket $H_\mathrm{S}[h_\mathrm{S}(id(s), id(p), id(o))]$. Similarly for triple patterns of type $(?, !, ?)$, $(?, ?, !)$, $(!, !, ?)$, $(!, ?, !)$ and $(?, !, !)$. However, since we do not assume collision-free hash functions, the relevant bucket may also contain references to triples that do not match the corresponding triple pattern. Hence, each triple referenced in a relevant bucket must still be checked. For triple patterns of type $(!, !, !)$ we may access any one of the hash tables; for instance, bucket $H_\mathrm{SO}[h_\mathrm{SO}(id(s), id(p), id(o))]$; and for triple patterns of type $(?, ?, ?)$ we have to inspect all n buckets in one of the hash tables. Table 1 summarizes what buckets are relevant for which kind of triple patterns.

Based on the presented index we propose three strategies for storing a query-local dataset: individual indexing, combined indexing, and quad indexing.

5.2 Individual Indexing

The main idea of the *individually indexed representation (IndIR)* of the query-local dataset is to keep the triples of each descriptor object in a separate index. Due to this separation we need an additional mapping idx that associates each URL from which a descriptor object was retrieved with the index that contains the triples of that descriptor object. To reduce the amount of required memory all indexes share a common dictionary[6].

IndIR allows for a straightforward implementation of the four main operations add, remove, replace and find: Adding a newly retrieved descriptor object D to the query-local dataset is a matter of creating a new index I_D, indexing all triples $t \in D$ in I_D, and adding the association $idx(url(D)) = I_D$. After this add operation finished successfully, it is not necessary to keep the added descriptor object anymore because it is completely represented by the index. To remove the descriptor object that was retrieved from

[6]We assume that the dictionary always assigns different identifiers to the blank nodes in different descriptor objects.

URL u from the query-local dataset, the association $idx(u) = I_D$ has to be removed and I_D has to be deleted. Replacing an old descriptor object that was retrieved from URL u with a new descriptor object D_{new}, retrieved from the same URL (i.e. $url(D_{\text{new}}) = u$), requires creating an index I_{new}, adding all $t \in D_{\text{new}}$ to I_{new}, atomically changing the association $idx(u) = I_{\text{old}}$ to $idx(u) = I_{\text{new}}$, and deleting I_{old}. To find matching triples in the query-local dataset $\mathcal{D}$, we have to inspect the relevant hash bucket in each index $I_D \in \{idx(url(D')) \mid D' \in \mathcal{D}\}$. However, since each matching triple that occurs in multiple descriptor objects must be reported only once, we have to record all reported triples in a temporary set; if we find a recorded triple again in the index of another descriptor object, we omit reporting it again.

These implementations of the modification operations add, remove and replace guarantee our isolation requirement naturally, as long as the modifications of idx are performed atomically. To guarantee the isolation requirement for the implementation of the find operation it is necessary to use a temporary, immutable copy of idx instead of iterating over the shared idx that might be modified by concurrent operations.

While the separation of the descriptor objects into individual indexes allows for a straightforward implementation, it introduces the potential for performance issues during the execution of the find operation. The need to access $|\mathcal{D}|$ different indexes and to iterate over multiple hash buckets may have a significant impact on the execution time, in particular for query-local datasets that contain a large number of descriptor objects. The combined indexing strategy addresses this issue.

5.3 Combined Indexing

The *combined index representation (CombIR)* of query-local datasets uses only a single index that contains the triples of all descriptor objects in its 6 hash tables. For CombIR the find operation can be implemented very efficiently: We only have to search through the references in the relevant hash bucket(s) of our single index.

Implementing the other three operations for CombIR is not as straightforward as it is for IndIR: Just adding the RDF triples from each descriptor object to the combined index would make it impossible to remove or replace a specific descriptor object because it would be unclear which of the indexed triples are part of it and have to be removed from the index. To enable the implementation of add, remove and replace in a way that guarantees our isolation requirement and that satisfies the efficiency requirements we adopt the idea of a multiversion database system [4]. In particular, we propose to use three additional mappings: src, $status$ and cur.

The mapping src associates each ID-encoded triple in the index with a set of unique identifiers that refer to the descriptor objects from which the triple originates. The mapping $status$ associates each of these identifiers with the current indexing status of the corresponding descriptor object. We distinguish the following *indexing status*es:

- *BeingIndexed* – This status indicates that the descriptor object is currently being added to the index.
- *Indexed* – This status indicates that the descriptor object is completely contained in the index and can be used by the find operation.
- *ToBeRemoved* – This status indicates that the descriptor object is completely contained in the index but it cannot be used by the find operation because the descriptor object has been removed from the query-local dataset or the query-local dataset already contains a more recent version of the descriptor object.
- *BeingRemoved* – This status indicates that the descriptor object is currently being removed from the index.

The mapping cur associates the URL of each descriptor object which is currently part of the query-local dataset with the identifier that refers to that descriptor object. For each URL u that is mapped by cur it must hold: $status(cur(u)) =$ *Indexed*.

Using the combined index and the mappings src, $status$ and cur we can implement the required operations as follows: To find matching triples in the query-local dataset $\mathcal{D}$, we have to search through the references in the relevant hash bucket of our combined index, but we ignore all ID-encoded triples $\bar{t}$ for which it does not hold: $\exists i \in src(\bar{t}) : status(i) =$ *Indexed*.

To add a descriptor object D to the query-local dataset we, first, have to create a unique identifier i for it and add the association $status(i) =$ *BeingIndexed*. For each RDF triple $t \in D$ we have to find $enc(t)$ in the index. If $enc(t)$ is not in the index, we add the association $src(enc(t)) = \{i\}$ and insert $enc(t)$ in the index; otherwise, we just add i to $src(enc(t))$. Finally, we change $status(i) =$ *BeingIndexed* to $status(i) =$ *Indexed* and add the mapping $cur(url(D)) = i$.

To remove the descriptor object that was retrieved from URL u from the query-local dataset, we change $status(cur(u)) =$ *Indexed* to $status(cur(u)) =$ *ToBeRemoved* and remove the association $cur(u) = i$. After these simple changes we can understand the descriptor object to be removed because it is ignored by find operations, even if this data is still present in our index. To delete this data completely we propose an additional operation clear that an asynchronous process executes regularly. This operation, first, changes all associations $status(i) =$ *ToBeRemoved* to $status(i) =$ *BeingRemoved*. Next, it searches all buckets from hash table H_{S} in our combined index for ID-encoded triples $\bar{t}$ with $\forall i \in src(\bar{t}) : status(i) =$ *BeingRemoved*. It removes the references to these triples from all 6 hash tables and deletes the corresponding associations from mapping src. Finally, it removes all associations $status(i) =$ *BeingRemoved*.

Replacing an old descriptor object that was retrieved from URL u with a new descriptor object D_{new} corresponds to such an execution of add for D_{new} that does not modify the mapping cur in the end. Instead, we complete the replacement by changing $status(cur(u)) =$ *Indexed* to $status(cur(u)) =$ *ToBeRemoved* and $cur(u) = i_{\text{old}}$ to $cur(u) = i_{\text{new}}$ where identifier i_{new} refers to D_{new}.

Due to the association of descriptor objects with indexing statuses we ensure the isolation requirement for the modification operations add, remove and replace. To guarantee isolation for the find operation it is necessary to atomically create a temporary copy of relevant buckets and inspect these immutable snapshots instead of iterating over the original buckets that might be modified by concurrent operations.

5.4 Quad Indexing

For CombIR we propose to use the mapping src to keep track of the origins of the triples in the combined index. An alternative strategy is the application of the concept of quads, or "*triples in context*" as Harth and Decker call them [9]. The *combined quad index representation (CombQuadIR)* implements this idea by using a slightly modified version of the index structure that we present in Section 5.1. Instead of the triple list, this modified version comprises a list of quads which we formally represent as pairs $(\bar{t}, i)$ where $\bar{t}$ is an ID-encoded triple and i is an identifier for a descriptor object. Consequently, the 6 hash tables in the modified version of the index structure do not keep references to ID-encoded triples but to the quads in the quad list. However, the hash functions of the index operate on the triple part of the quads only, exactly as described for the original version of the index structure. Hence, the relevancy of hash buckets for triple patterns as specified in Table 1 still holds.

While the use of quads obsoletes the mapping src, CombQuadIR still requires the mappings $status$ and cur that we introduced for CombIR. The implementation of the four required operations for the quad index is also similar to their implementation for CombIR: To add a descriptor object D to the query-local dataset we create a unique identifier i and add the association $status(i) =$ *BeingIndexed*. For each RDF triple $t \in D$ we insert a new quad $(enc(t), i)$ into the index. Finally, we change $status(i) =$ *BeingIndexed* to $status(i) =$ *Indexed* and add the mapping $cur(url(D)) = i$.

To remove or replace descriptor objects we can use the same implementations as introduced for CombIR. However, the asynchronously executed clear operation has to be adjusted: After changing all associations $status(i) =$ *ToBeRemoved* to $status(i) =$ *BeingRemoved* it goes through all quads in the quad list of the index. For each quad $q = (\bar{t}, i)$ with $status(i) =$ *BeingRemoved* it removes the references to q from all 6 hash tables and deletes q in the quad list. Finally, it removes all associations $status(i) =$ *BeingRemoved*.

For the find operation we have to search through the references in the relevant hash bucket of the combined quad index, but we ignore all quads $(\bar{t}, i)$ for which $status(i) \neq$ *Indexed*. To ensure that we report each matching triple only once, even if it occurs in multiple descriptor objects and, thus, is part of multiple quads, we apply the same strategy as we use for the find operation of IndIR: We record reported triples temporary and do not report them again if they are part of another quad referenced by the relevant hash bucket.

For the same reasons as discussed for CombIR, it is possible to guarantee that an implementation of add, remove, replace and find for CombQuadIR satisfies our isolation requirement.

6. EVALUATION

In this section we analyze and compare IndIR, CombIR, and CombQuadIR empirically. We describe the experimental setup, determine suitable parameters for the indexes that we use in these data structures, and compare the data structures with each other as well as with similar data structures available in existing systems.

6.1 Experimental Environment

We implemented the data structures and the corresponding operations as introduced in this paper. For the indexes we use hash tables with a number of n buckets where $n = 2^m$ for some exponent m and hash functions that return the m least significant bits of the term identifiers at specific positions in ID-encoded triples[7]:

$$h_S(\bar{s}, \bar{p}, \bar{o}) \equiv \bar{s} \oplus \text{bitmask}^{[m]}$$
$$h_P(\bar{s}, \bar{p}, \bar{o}) \equiv \bar{p} \oplus \text{bitmask}^{[m]}$$
$$h_O(\bar{s}, \bar{p}, \bar{o}) \equiv \bar{o} \oplus \text{bitmask}^{[m]}$$
$$h_{SP}(\bar{s}, \bar{p}, \bar{o}) \equiv (\bar{s} \cdot \bar{p}) \oplus \text{bitmask}^{[m]}$$
$$h_{SO}(\bar{s}, \bar{p}, \bar{o}) \equiv (\bar{s} \cdot \bar{o}) \oplus \text{bitmask}^{[m]}$$
$$h_{PO}(\bar{s}, \bar{p}, \bar{o}) \equiv (\bar{p} \cdot \bar{o}) \oplus \text{bitmask}^{[m]}$$

We use these hash functions because they can be calculated very efficiently. Furthermore, the calculated hash values are distributed sufficiently because our dictionary implementation does not calculate term identifiers based on some representation of the RDF terms; instead, we use a counter that provides a new identifier whenever the id function is called for an unknown RDF term.

Our implementation is written in Java and available as Free Software as part of SQUIN[8], our link traversal based query execution system. To evaluate the query performance provided by the three data structures we used the query engine in SQUIN. However, for most experiments we disabled the traversal of data links during the query execution and used pre-populated query-local datasets instead. This strategy allowed us to avoid measuring the effects of URI look-ups and data retrieval.

For our experiments we adopt the Berlin SPARQL Benchmark (BSBM) [6]. The BSBM executes different mixes of 25 SPARQL queries over synthetic RDF data. This data describes entities in a distributed e-commerce scenario, including different producers and vendors, the offered products, and reviews from multiple reviewing systems. The data is generated automatically and it is possible to use datasets that vary in the number of described entities and, thus, the amount of RDF triples. We use these BSBM datasets to prepare multiple query-local datasets of different sizes for our experiments: We understand each dataset generated for BSBM as the union of multiple datasets that could be exposed as Linked Data on the Web. Thus, looking up the URI of any of the described entities would result in a descriptor object which contains only these RDF triples from the BSBM dataset that describe the corresponding entity. To generate a query-local dataset that contains all these descriptor objects we query a given BSBM dataset with SPARQL DESCRIBE queries for all relevant entities in that dataset; the result of each DESCRIBE query becomes a descriptor object in the query-local dataset that we generate[9]. Table 2 describes the query-local datasets that we prepared for the experiments. These datasets are typical for the query-local datasets that SQUIN usually processes.

Table 2: Statistics of query-local datasets for our experiments.

q.-local dataset	BSBM scaling factor	Number of descriptor objects	Overall number of RDF triples
$\mathcal{D}_{50}$	50	2,599	22,616
$\mathcal{D}_{100}$	100	4,178	40,133
$\mathcal{D}_{150}$	150	5,756	57,524
$\mathcal{D}_{200}$	200	7,329	75,062
$\mathcal{D}_{250}$	250	9,873	97,613
$\mathcal{D}_{300}$	300	11,455	115,217
$\mathcal{D}_{350}$	350	13,954	137,567
$\mathcal{D}_{500}$	500	18,687	190,502

In addition to merely comparing our implementations of IndIR, CombIR, and CombQuadIR among each other, we also compare these implementations with existing data structures that can be used to store a query-local dataset: the main memory implementation of the `NamedGraphSet` interface in NG4J[10] and the main memory implementation of the `DatasetGraph` interface in ARQ[11], the query engine for the Jena framework. Both implementations store each descriptor object in a separate data structure, similar to our individually indexed representation IndIR. However, while we use indexes with ID-encoded triples, NG4J and ARQ use an implementation of the Jena `Graph` interface that represents RDF terms

[7]With $\oplus$ we denote the bitwise AND operator and bitmask$^{[m]}$ is a bitmask in which the m least significant bits are set.

[8]http://squin.org

[9]For a more detailed description and the code of the corresponding extension for the BSBM data generator we refer to a blog post at http://sourceforge.net/apps/wordpress/squin/2009/04/15/a-data-generator-for-bsbm-that-provides-linked-data-characteristics/

[10]http://www4.wiwiss.fu-berlin.de/bizer/ng4j/

[11]http://openjena.org/ARQ/

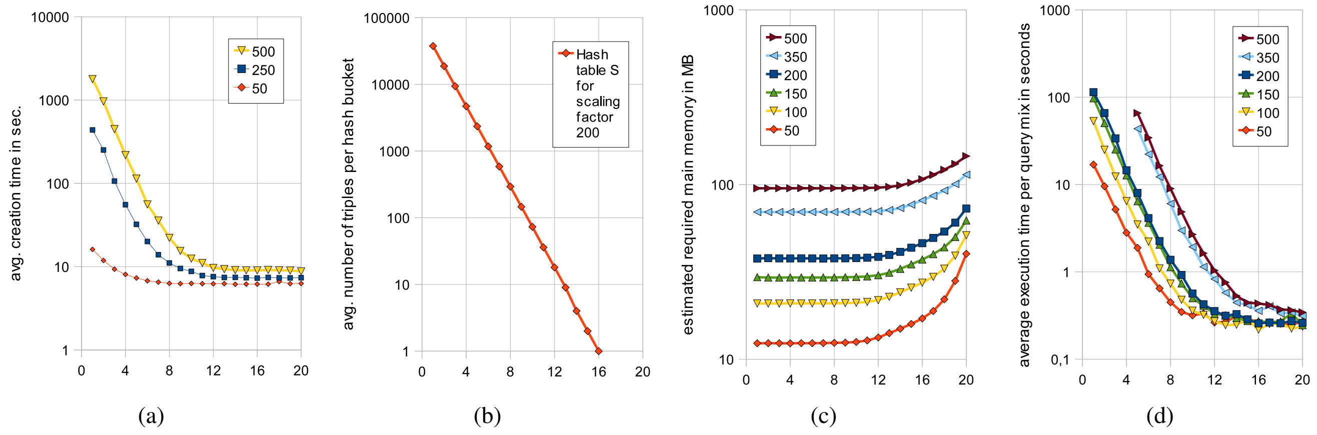

Figure 1: Measurements for combined index representations of different query-local datasets with varying m (from 1 to 20): (a) average creation times for representations of $\mathcal{D}_{50}$, $\mathcal{D}_{250}$ and $\mathcal{D}_{500}$, (b) average number of triples per hash bucket in H_S after loading $\mathcal{D}_{200}$, (c) estimated memory consumed by the representations, and (d) average time to execute BSBM query mixes over the representations.

as Java objects. For the experiments we used the latest releases of Jena (v.2.6.4) and ARQ (v.2.8.7) and a recent version of NG4J (CVS check-out from Jan. 20, 2011).

We conducted all experiments on 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 and it runs a recent 32 bit version of Gentoo Linux with Sun Java 1.6.0. All software used for this evaluation, including the scripts to run the experiments, as well as all our measurements can be found on the Website for this paper[12].

6.2 Tuning the Combined Index

Before we compared IndIR, CombIR, and CombQuadIR we had to determine suitable parameters for the indexes that we use in these data structures. In the following we discuss the reasons for this need and describe the analysis based on which we tuned the indexes.

The number of buckets per hash table has an impact on the performance of find operations and, thus, query execution times, because a higher number corresponds to less elements per buckets and it is faster to search through a smaller number of potentially matching triples. Furthermore, the number of buckets may also affect the amount of main memory required for the indexes if the implementation of the buckets always allocates some spare memory so that buckets are never filled to their full capacity. For these reasons it is necessary to decide on a number of buckets that is suitable for the typical amount of RDF data which has to be stored in the hash tables. Since this amount is different for the individual indexes and the combined index we require a different number of buckets for these indexes. As we describe in Section 5.1, the hash tables in our index structure contain $n = 2^m$ buckets. Thus, to provide for a fair comparison of IndIR, CombIR, and CombQuadIR we have to find suitable exponents m for the indexes that we use in these data structures. In this paper we focus on tuning CombIR. However, we conducted similar analyses for the other two data structures.

As outlined before, selecting a suitable number of buckets (or m) is a trade-off between query performance and required memory. To find a suitable m for the combined index we used different m, varying from 1 to 20, to create different combined index representations of the aforementioned query-local datasets (cf. Table 2).

[12]http://squin.org/experiments/IndexStructuresLDOW2011/

For each of these representations we measure the creation time, the required memory, and the query performance.

To determine the *creation times* we loaded our query-local datasets into an ARQ `DatasetGraph` and measured the time to create combined index representations from it. For each m we followed this procedure 5 times and took the mean of the results. The chart in Figure 1(a) illustrates these measurements for the different combined index representations of the query-local datasets $\mathcal{D}_{50}$, $\mathcal{D}_{250}$ and $\mathcal{D}_{500}$. As can be seen from this chart, the creation times are small for representations with a high m (e.g. about 6.3 s and 8.8 s for the $m = 20$ representations of $\mathcal{D}_{50}$ and $\mathcal{D}_{500}$, respectively) and they increase significantly with a decreasing m. The reason for this behavior is the number of elements that have to be added per hash bucket: For a high m we have a large number of buckets in the hash tables so that each bucket is only relevant for a few triples as Figure 1(b) illustrates. In this case, the time to encode all RDF triples, find the relevant hash buckets for each, and add the corresponding references into these buckets is roughly constant, independent of m. However, for a small number of buckets per hash table (i.e. for a small m), adding the references to the buckets is more expensive and impossible in constant time because each bucket is relevant for a much greater number of triples (cf. Figure 1(b)). Thus, for a decreasing m we reach a point where the time to fill the buckets dominates the overall creation time.

To measure the *required memory* we applied a method for Java object profiling as proposed in [18]. Using this method we estimated the amount of main memory consumed by the different combined index representations of the query-local datasets. Figure 1(c) illustrates the results for $\mathcal{D}_{50}$, $\mathcal{D}_{100}$, $\mathcal{D}_{150}$, $\mathcal{D}_{200}$, $\mathcal{D}_{350}$ and $\mathcal{D}_{500}$; while the x-axis denotes the exponents m, the y-axis denotes the estimated amount of memory (in MB) consumed by the different representations. For each dataset we note that representations with a smaller number of buckets per hash table (i.e. a smaller m) require roughly the same amount of memory (e.g. about 12.5 MB for $\mathcal{D}_{50}$ and about 95.3 MB for $\mathcal{D}_{500}$). For representations with an m greater than 10 the required memory starts to increase exponentially. We explain this behavior as follows: For smaller m the required memory is dominated by a constant amount of memory required for the dictionary and the ID-encoded triples, which is independent of m for each dataset. At some point, $m > 10$ in our experiments, this constant amount is largely exceeded by the memory required for the hash buckets that are not filled to their ca-

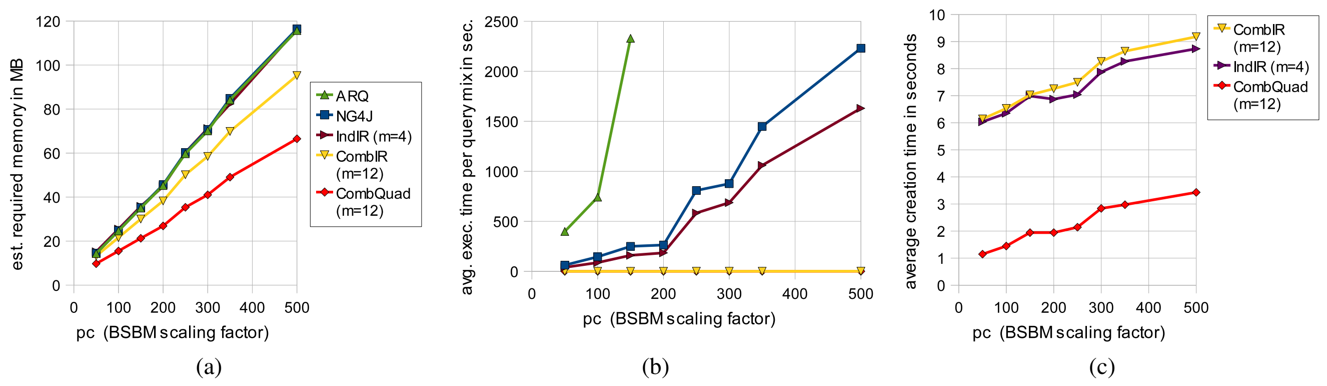

Figure 2: Measurements for different data structures storing the query-local datasets $\mathcal{D}_{pc}$ from Table 2: (a) estimated memory consumed by the data structures, (b) average time to execute BSBM query mixes over the data structures, and (c) average creation times for the data structures.

pacity. The exponential growth can be attributed to the exponential correlation between the number of buckets per hash table and m.

To measure the *query performance* provided by CombIR initialized with different m we used the different combined index representations of each of our query-local datasets and executed identical BSBM (V2.0) query mixes over them. Hence, for each representation we run the BSBM query mix 13 times where the first 3 runs were for warm up and are not considered for the measurements. For the actual execution of queries we used the query engine in SQUIN. However, we disabled the look-up of URIs for this experiment in order to actually measure the efficiency of the find operations and to avoid distortions caused by the effects of network access, data retrieval, etc. Figure 1(d) depicts the average times to execute the query mixes for CombIR with different m. For all query-local datasets, the execution times are small for a high m (e.g. 0.28 s and 0.44 s for $\mathcal{D}_{50}$ and $\mathcal{D}_{500}$ with $m = 15$, respectively) and they increase significantly with a decreasing m. Similarly to the behavior of the creation time, this observation can be attributed to the notable differences in the average number of triples per hash bucket: While the time to find relevant buckets is constant, searching through the entries in these buckets and checking them w.r.t. a given triple pattern gets more expensive for a small number of buckets that reference more triples each.

Based on the results of these experiments we identified a value of 12 for m as the best trade-off between required memory and query performance (incl. creation time) for CombIR. Note, for applications that operate on much smaller or much larger query-local datasets than those that we used for the experiments, we suggest to find a value for m that is more suitable in these scenarios. However, in the remainder of this paper we use $m = 12$ for the combined index. To tune CombQuadIR accordingly we conducted the same experiments and found that $m = 12$ is also the best choice for the quad index. Similarly, we analyzed IndIR. According to this analysis[13] $m = 4$ is most suitable for the individual indexes that contain a single descriptor object each.

6.3 Comparing the Data Structures

We use our implementations of IndIR, CombIR, and CombQuadIR to empirically compare the three approaches for storing a query-local dataset. Furthermore, we also compare our implementations with the main memory implementation of similar data structures in NG4J and ARQ as introduced before (cf. Section 6.1). For our comparison we conducted the same kind of experiments as in the previous section: We measured the required memory, the query execution time, and the creation time. Figures 2(a) to 2(c) illustrate the results of these experiments. In the following we discuss the results of these experiments.

The chart in Figure 2(a) illustrates the estimated amount of memory that was required to store the query-local datasets $\mathcal{D}_{50}$ to $\mathcal{D}_{500}$ using the different data structures. As can be seen from this chart, CombQuadIR required less memory than the other data structures from which CombIR required slightly less than ARQ, NG4J and IndIR. We attribute the latter observation to the fact that ARQ, NG4J and IndIR have to allocate memory for Java objects that represent the containers of the separated descriptor objects. For instance, our implementation of IndIR has to provide a separate Java object for each of the individual indexes, whereas, CombIR and CombQuadIR require only one of these objects. The other observation is the advantage of CombQuadIR over CombIR w.r.t. memory consumption. This observation shows that storing the src associations for each indexed triple requires more space in our implementation than storing multiple quads that contain the same triple.

While the amount of memory required for the different data structures was at least in the same order of magnitude, the provided query performance differed significantly: In comparison to ARQ, NG4J and IndIR, we measured almost constant query mix execution times for CombIR and CombQuadIR (cf. Figure 2(b)). Even if these times increase slightly with the size of the query-local dataset (e.g. from an average of 0.3 seconds to execute a BSBM query mix for CombIR of $\mathcal{D}_{50}$ to an average of 1.3s for $\mathcal{D}_{500}$), we consider this increase insignificant to the increase we measured for the other data structures. The reason for these remarkable differences is the number of indexes that have to be accessed during the execution of find operations: While it is only a single index for CombIR and CombQuadIR, independent of the number of descriptor objects in the query-local dataset, IndIR needs as many index accesses as descriptor objects are contained in the query-local dataset; similarly for NG4J and ARQ. Figure 2(b) also illustrates that ARQ performed very badly (for the larger datasets query execution aborts with a stack overflow error) and that IndIR was slightly better than the NG4J data structure. We explain the latter by our use of numerical identifiers to represent RDF terms; these identifiers are faster to process and compare than the Java objects used by NG4J/Jena. We also note that CombIR and CombQuadIR performed equally well.

[13]http://sourceforge.net/apps/wordpress/squin/2009/04/25/identified-a-proper-index-size-for-the-new-swcllib-storage-solution/

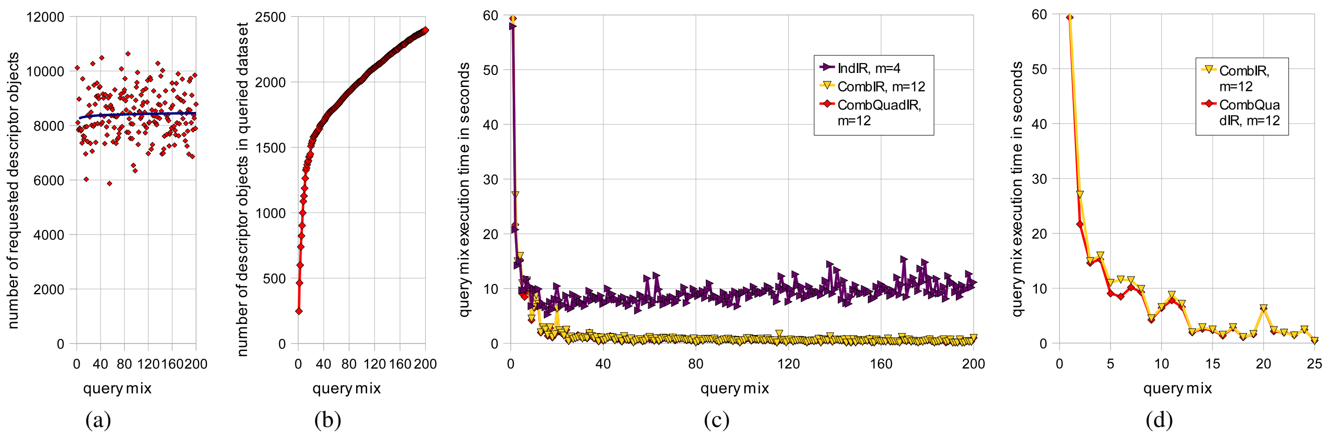

Figure 3: Measurements for a link traversal based execution of a sequence of 200 BSBM query mixes that re-use the local dataset.

Finally, we measured the creation times[14] for our implementations of IndIR, CombIR, and CombQuadIR. Figure 2(c) denotes the results of this experiment: The creation times for our implementation of CombQuadIR are much smaller than for IndIR and CombIR. In particular, the difference between CombIR and CombQuadIR might be surprising given the similarity of both data structures. However, the need to check whether the combined index already contains each triple that has to be inserted is the reason why CombIR has higher creation times than CombQuadIR for which inserting a triple just means adding a new quad. Focusing solely on IndIR and CombIR we note that the creation times for our implementations of these data structures are comparable, even if the individually indexed representations were created slightly faster. However, this small advantage of IndIR over CombIR does not outweigh the divergence that we measured for the query execution times.

6.4 Evaluating the Impact on Link Traversal Based Query Execution

In all experiments discussed so far, we used query-local datasets that we populated in advance. This strategy allowed us to avoid measuring the effect of link traversal and network access, which affect the overall execution time of link traversal based query execution. Hence, we were able to evaluate the actual query performance provided by the different data structures. This evaluation revealed a clear advantage of CombIR and CombQuadIR over the other data structures, where CombQuadIR is superior to CombIR due to the smaller creation times. A question that remains is whether these differences have an impact on the overall execution time of link traversal based query execution. To answer this question we enabled the traversal of data links in our query engine and conducted an experiment that we describe in the remainder of this section.

Using the BSBM dataset generated with a scaling factor of 50 (cf. Table 2) we set up a Linked Data server[15] which publishes the generated data following the Linked Data principles. With this server we simulate the Web of Data in order to conduct the experiment in a controlled environment. To query over this simulated Web we adjusted the BSBM queries in such a way that they access our simulation server. For each data structure we ran the same sequence of 200 BSBM query mixes, using a single, initially empty query-local dataset. Hence, we did not clear the query-local dataset between the queries so that each query execution could benefit from the local availability of all descriptor objects that have been retrieved for previous queries. For each of the 200 query mixes we measured the overall query execution time and the number of descriptor objects that were contained in the query-local dataset after executing the query mix.

Figure 3(a) denotes the number of descriptor objects that had been requested during the execution of queries in each of the 200 query mixes. The trendline in this chart illustrates that this number remained roughly constant for our sequence of query mixes. However, the number of newly discovered descriptor objects added to the query-local dataset decreased for later query mixes in the sequence as can been seen from the overall number of descriptor objects contained in the query-local dataset after each query mix (cf. Figure 3(b)). Hence, many descriptor objects that were requested during the execution of queries in later query mixes had already been retrieved during the execution of previous queries. This observation indicates that certain queries in the whole sequence of query mixes are similar w.r.t. the data that is required to answer them. Applications that generate a query workload with similar characteristics are most likely to benefit from re-using the query-local dataset for the execution of multiple queries.

Figure 3(c) illustrates the overall execution time of each of the 200 query mixes for the different data structures. We note that all three data structures exhibit the same general behavior: The first query mixes have a significantly higher execution time than later query mixes. We explain this behavior as follows: For the first query mixes the query execution times are dominated by the effects of link traversal (e.g. the need to retrieve descriptor objects from the Web and to add them to the data structure that stores the query-local dataset). However, with a decreasing number of descriptor objects that have to be retrieved and added, this domination decreases, until, at around the 13th query mix in our experiment, the trend changes completely. An additional indicator for this domination is the 20th query mix, for which an untypical high number of 60 additional descriptor objects were discovered (in comparison, for the query mixes #19 and #21 it was 15 and 19, respectively). Due to the retrieval of this comparably high number of newly discovered descriptor objects the query execution times measured for all three data structures peak again.

After the domination of the effects of link traversal disappeared (i.e. after query mix #20) the three data structures exhibit the same

[14] As mentioned before, we understand *creation time* as the time to create our data structures from an ARQ `DatasetGraph` object into which we loaded the corresponding dataset before. Hence, based on this understanding ARQ has a creation time of 0 seconds.

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

behavior w.r.t. query execution times as they did in the previous experiments without link traversal based query execution: For CombIR and CombQuadIR the query engine required roughly the same execution time for all query mixes, independent of the still increasing size of the query-local dataset. For IndIR, in contrast, the execution times are higher and they begin to increase slightly. We expect that this increase could even rise, given that the query-local dataset contained 2.397 descriptor objects after query mix #200, which is comparable to $\mathcal{D}_{50}$ used in the previous experiments (cf. Table 2). As a final observation we note that the smaller creation times of CombQuadIR impact the overall execution times only when a significant number of newly discovered descriptor objects are added during link traversal based query execution. Figure 3(d) provides a detailed view on the query execution times for query mixes #1 to #25 to illustrate this effect: For the first query mixes (during which more descriptor objects were retrieved) we measured slighly smaller query execution times for CombQuadIR than for CombIR. However, with a decreasing number of newly discovered descriptor objects, the advantage of CombQuadIR over CombIR vanishes.

7. CONCLUSION

In this paper we investigate main memory data structures to store a temporary collection of Linked Data from the Web as is required in a link traversal based query execution system. We discuss that these data structures must support efficient, multi-threaded loading and triple-based querying of many small sets of RDF data, which is a requirement that is not sufficiently addressed by existing work. Therefore, we introduce and analyze three alternative data structures that are suitable in our usage scenario. These data structures make use of hash table based indexes and present i) an individually indexed organization of the data, ii) a combined index, and iii) a combined quad index. In an empirical evaluation we demonstrate a significant performance improvement that can be achieved by using the combined index or the combined quad index. Comparing these two alternatives, our evaluation revealed that the quad index features smaller creation times and, thus, is advantageous in cases where new data is retrieved and added frequently. As future work we aim to investigate whether we can improve the query performance or memory requirements of the data structures by using another hash function for the indexes.

8. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 2009.

[2] M. Atre, J. Srinivasan, and J. A. Hendler. BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries. In *Proceedings of the Poster and Demo Session at the 7th International Semantic Web Conference (ISWC)*, 2008.

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

[4] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control – Theory and Algorithms. *ACM Transactions on Database Systems*, 8, 1983.

[5] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. SpiderStore: Exploiting Main Memory for Efficient RDF Graph Representation and Fast Querying. In *Proceedings of the Workshop on Semantic Data Management (SemData) at VLDB*, 2010.

[6] C. Bizer and A. Schultz. Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints. In *Proceedings of the Workshop on Scalable Semantic Web Knowledge Base Systems at ISWC*, 2008.

[7] P. Bouquet, C. Ghidini, and L. Serafini. Querying The Web Of Data: A Formal Approach. In *Proceedings of the 4th Asian Semantic Web Conference (ASWC)*, 2009.

[8] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15, 1983.

[9] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the 3rd Latin American Web Congress (LA-Web)*, 2005.

[10] 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.

[11] O. Hartig. How caching improves efficiency and result completeness for querying linked data. In *Proceedings of the 4th International Linked Data on the Web workshop (LDOW) at WWW*, 2011.

[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] M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store And High Performance Memory System for Semantic Association Discovery. In *Proceedings of the 4th International Semantic Web Conference (ISWC)*, 2005.

[15] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, 2008.

[16] E. Oren, C. Gueret, and S. Schlobach. Anytime Query Answering in RDF through Evolutionary Algorithms. In *Proceedings of the 7th International Semantic Web Conference (ISWC)*, 2008.

[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] V. Roubtsov. Sizeof for Java. http://www.javaworld.com/javaworld/ javaqa/ 2003-12/ 02-qa-1226-sizeof.html, 2003.

[19] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. *Proceedings of the VLDB Endowment*, 1, 2008.

[20] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, 2004.

[21] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *Proceedings of the 4th International Semantic Web Conference (ISWC)*, 2005.

[22] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A Graph Based RDF Index. In *Proceedings of the 21nd AAAI Conference on Artificial Intelligence (AAAI)*, 2007.

[23] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, 2008.