DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

On Lightweight Data Summaries for Optimised Query Processing over Linked Data

Andreas Harth Katja Hose Marcel Karnstedt
Axel Polleres Kai-Uwe Sattler Jürgen Umbrich

DERI Technical Report 2009-11-17
November 2009

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

DERI Galway
IDA Business Park
Lower Dangan
Galway, Ireland
http://www.deri.ie/

DERI Technical Report

DERI Technical Report 2009-11-17, November 2009

On Lightweight Data Summaries for Optimised Query Processing over Linked Data

Andreas Harth [1] Katja Hose [2] Marcel Karnstedt [3] Axel Polleres [4]
Kai-Uwe Sattler [5] Jürgen Umbrich [6]

Abstract. Typical approaches for querying structured Web Data collect (crawl) and pre-process (index) large amounts of data in a central data repository before allowing for query answering. This time-consuming pre-processing phase however leverages the benefits of Linked Data – where structured data is accessible live and up-to-date at distributed Web resources that may change constantly – only to a limited degree, as query results can never be up-to-date. An ideal query answering system for Linked Data should return current answers in a reasonable amount of time, even on corpora as large as the Web. Query processors evaluating queries directly on the live sources require knowledge of the contents of data sources. In this paper, we develop and evaluate an approximate index structure summarising graph-structured content of sources adhering to Linked Data principles, provide an algorithm for answering conjunctive queries over Linked Data on the Web exploiting the source summary, and evaluate the system using synthetically generated queries. The experimental results show that our lightweight index structure enables complete and up-to-date query results over Linked Data, while keeping the overhead for querying low and providing a satisfying source ranking "for free".

Keywords: On-demand, Web of Data, Data Summary, Conjunctive Queries.

[1]AIFB, Karlsruhe Institute of Technology, Germany. harth@kit.edu
[2]Max-Planck Institute for Informatics, Saarbrücken, Germany. hose@mpi-inf.mpg.de
[3]DERI, National University of Ireland, Galway, Ireland. marcel.karnstedt@deri.org
[4]DERI, National University of Ireland, Galway, Ireland. juergen.umbrich@deri.org
[5]Ilmenau University of Technology, Ilmenau, Germany. kus@tu-ilmenau.de
[6]DERI, National University of Ireland, Galway, Ireland. juergen.umbrich@deri.org

Acknowledgements: This material is based upon works in parts jointly supported by the Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2) and under Grant No. 08/SRC/I1407 (Clique: Graph & Network Analysis Cluster).

Contents

1 Introduction **1**

2 Querying Linked Data **3**

3 Source Selection using Data Summaries **5**
3.1 Source Indexing using the QTree 5
3.2 Source Selection 7
3.2.1 Triple Pattern Source Selection 7
3.2.2 Join Source Selection 8
3.3 Source Ranking 10

4 Data Summary Construction & Maintenance **11**
4.1 Initial Phase 11
4.2 Expansion Phase 12

5 Evaluation **13**
5.1 Setup 13
5.2 Results 14
5.2.1 Quality of Source Selection 14
5.2.2 Impact of Ranking 14
5.2.3 Query Execution Time 15
5.2.4 Comparison with Other Approaches 16
5.3 Discussion 17

6 Related Work **18**

7 Conclusion & Future Work **18**

1 Introduction

The recent developments around Linked Data promise to lead to the exposure of large amounts of data on the Semantic Web amenable to automated processing in software programs [1]. Linked Data sources use RDF (Resource Description Format) in various serialisation syntaxes for encoding graph-structured data. The Linked Data effort is part of a trend towards highly distributed systems, with thousands or potentially millions of independent sources providing small amounts of structured data. Using the available data in data integration and decision-making scenarios requires query processing over the combined data.

For evaluating queries in such environments we can distinguish two directions:

- data warehousing or *materialisation-based approaches (MAT)*, which collect the data from all known sources in advance, preprocess the combined data, and store the results in a central database; queries are evaluated using the local database.

- *distributed query processing approaches (DQP)*, which parse, normalise, and split the query into subqueries, determine the sources containing results for subqueries, and evaluate the subqueries against the sources directly.

Unfortunately, applying DQP directly is not a viable for Linked Data sets: Firstly, in most cases the data in the different sources cannot be described by simple expressions because they may vary in the schema or do not even have common values. Secondly, queries cannot be "dispatched", unless query processing capabilities exist at the source sites. Preliminary results for distributed query processing over distributed RDF sources [26] assume, similar to resp. approaches from the traditional database works, relatively few query endpoints with probably huge amounts of data, rather than many small Web resources accessible via simple HTTP GET only.

The aim of the present paper is to narrow the gap between these two extreme approaches and find a reasonable middle-ground for processing queries over Linked Data sources directly. Although currently only a few data sources offer full query processing capabilities (e.g., by implementing SPARQL [4,25], a query language and protocol for RDF), we still can eschew the cost of maintaining a full index of the data at a central location. On the current Web, all we can assume is that the sources implement a single operation *GET* which returns the content of the source in RDF. Thus, instead of full federation, in this work we propose an approximate multidimensional indexing structure (a QTree [15]) to store descriptions of the content of data sources. The QTree forms the basis for sophisticated query optimisation and helps the query processor decide on which sources a query or a subquery to route to. We assume – as typical for Linked Data – a large number of sources, which, in contrast to typical data integration scenarios, are of small size in the range of a few kilobytes to megabytes.

Approximate data summaries such as QTrees can be populated by crawling techniques similar to those employed by centralised systems, with the advantage of a significantly smaller index, which can be kept in memory, and live query results, by processing the actual query only over those sources which likely contain relevant information. Also, such a QTree index can be dynamically extended, by adding either user-submitted sources or sources discovered during the query processing.

The strategy we propose is a reasonable compromise under the assumption that the overall data distribution does not change dramatically over time, that is, that the distribution characteristics are relatively stable, which holds for a wide range of Linked Data sources (e.g., DBpedia[1], DBLP[2], or machine-readable personal homepages. Under this assumption we can employ an approach which stores a data summary reflecting these immutable characteristics in lieu of a full local data index.

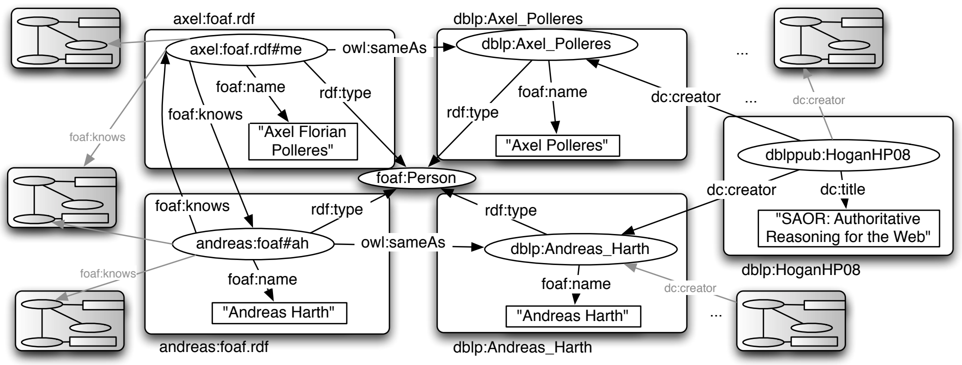


Figure 1: Linked Data in RDF about persons and their publications

Our approach works as follows:

- prime an approximate index structure (a QTree) with a seed data set (various mechanisms for creating and maintaining the index are covered in Section 4)
- use the QTree to determine which sources contribute partial results for a conjunctive SPARQL query Q
- fetch the content of the sources (optionally using only the top-k sources according to cardinality estimates stored in the QTree) into memory
- perform join processing locally given remote sources do not provide functionality for computing joins

The main problem of processing such queries hence becomes finding the right sources to contain possible answers that can contribute to the overall query and efficient parallel fetching of content from these sources.

We conclude the section by introduction example data and queries used throughout the paper. Section 2 discusses alternative methods for answering queries over Linked Data. In Section 3, we present an approach to select sources from a QTree, an approximate data summary index. Section 4 describes approaches to construct and maintain these data summaries followed by a discussion of results of an evaluation in Section 5. In Section 6, we align our system with existing work and conclude with an outlook to future work in Section 7.

Example As an example used throughout the paper, consider a scenario in which sources publish interlinked data about people, the relations between them and their publications. Such data is

[1]`http://dbpedia.org/`
[2]`http://dblp.l3s.de/d2r/`

indeed available as Linked Data in RDF on the Web in the form of hand-crafted files in the friend-of-a-friend (FOAF) vocabulary [2]) describing personal information as well as social connections, and automatic exports of publication databases such as DBLP.

For instance, consider the Linked Data sources depicted in Figure 1. RDF graphs comprise (`subject predicate object`) triples that denote labelled edges between the subject and the object. The figure shows five RDF graphs covering data about Andreas and Axel: personal homepages encoded in FOAF, data covering personal information at DBLP, and data about one of their joint publications at DBLP. We assume that `namespace:localname` pairs expand to full URIs, e.g. `dblp:Axel_Polleres` expands to `http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres`.

Conjunctive SPARQL queries[3] consist of so-called *basic graph patterns* (BGPs), i.e., sets of triple patterns containing variables. For instance, the following query asks for names of Andreas' friends:

```
SELECT ?n WHERE {
  andreas:foaf#ah foaf:knows ?f. ?f foaf:name ?n. }                    (1)
```

The next query asks for authors of article `dblppub:HoganHP08` who mutually know each other:

```
SELECT ?x1 ?x2 WHERE {
  dblppub:HoganHP08 dc:creator ?a1, ?a2.
  ?x1 owl:sameAs ?a1. ?x2 owl:sameAs ?a2.                              (2)
  ?x1 foaf:knows ?x2. ?x2 foaf:knows ?x1. }
```

2 Querying Linked Data

Linked Data [1] is RDF published on the Web according to the principles: 1) use URIs as names for things 2) use (dereferenceable) HTTP URIs, 3) provide useful content at these URIs (e.g. encoded in RDF), and 4) include links to other URIs for discovery. In the same way the current Web is formed by HTML documents and hyperlinks between documents, the Linked Data Web is constructed by using HTTP URIs (principle 1 and 2). Principle 3, providing meaningful content for dereferenced URIs (that is, RDF triples describing the URI, typically in the subject position) allows for a new way to perform lookups on the data during query runtime. The principle provides a correspondence (in URI syntax or via redirects in the HTTP protocol) between a URI of a resource and the data source. For example, the resource URI `http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres` redirects to the source URI `http://dblp.l3s.de/d2r/page/authors/Axel_Polleres`. Finally, reusing URIs across sources (principle 4) makes sure that data covering the same entity can be collated from multiple sources.

Most current approaches enabling query processing over RDF data operate very much along the lines of relational data warehouses or search engines; Semantic Web search engines [3, 6, 13, 22] crawl large amounts of RDF documents for materialisation and indexing in a centralised data store.

The centralised approaches using materialisation (MAT) provide excellent query response times due to the large amount of preprocessing carried out during the load and indexing steps, but suffers from a number of drawbacks. First, the aggregated data is never current as the process of collecting and indexing vast amounts of data is time-consuming. Second, from the viewpoint of a single requester with a particular query, there is a large amount of unnecessary data gathering,

[3]We focus on the core case of conjunctive queries and do not consider more complex features such as unions, outer joins, or filters available in SPARQL, which could be however layered on top of conjunctive query functionality.

processing, and storage involved since a large portion of the data might not be used for answering that particular query. Furthermore, due to the replicated data storage the data providers have to give up their sole sovereignty on their data (e.g., they cannot restrict or log access any more since queries are answered against a copy of the data).

On the other end of the spectrum, there are approaches that assume processing power attainable at the sources themselves (*DQP*), which could be leveraged in parallel for query processing. Such distributed or federated approaches [11] offer several advantages: the system is more dynamic with up-to-date data and new sources can be added easily without time lag for indexing and integrating the data, and the systems require less storage and processing resources at the query issuing site. The potential drawback, however, is that DQP systems cannot give strict guarantees about query performance since the integration system relies on a large number of potentially unreliable sources. DQP is a well-known database problem [18]. Typically, DQP involves the following steps for transforming a high-level query into an efficient query execution plan: parsing, rewriting by applying equivalence rules in order to normalise, unnest, and simplify the query, data localisation, optimisation (i.e., replacing the logical query operators by specific algorithms and access methods as well as by determining the order of execution both at global level as at the different local sites), and finally execution. Besides optimisation, data localisation is an important step that affects the efficiency of the execution. The goal of data localisation, also known as *source selection*, is to identify the source sites that possibly provide results for the given query or – in other words – to eliminate sites from the query plan which do not contribute to the result. In classic distributed databases this step is supported by (query or view) expressions describing the fragmentation of a global table.

Possible approaches to evaluate queries over such Web resources and particularly addressing the problem of source selection are:

- **Direct Lookups (DL)** The direct lookup approach is implemented in [10] where one tries to leverage the correspondence between source addresses and identifiers contained in the sources to answer queries. The query processor performs lookups on the sources which contain identifiers mentioned in the query or are retrieved in subsequent steps. To answer query (1), one could fetch content from `andreas:foaf#ah`, dereference `foaf:knows` links, and gather new information where hopefully the respective names of friends are found. The sources in the DBLP realm are irrelevant for answering this query. The strategy fails to find the solutions for query (2), however, since the necessary `owl:sameAs` links come from outside the linked closure of the graph `dblppub:HoganHP08`. Apart from possible incompleteness issues the approach also has limitations in the sense that only limited parallelisation is possible: the query processor starts with one source and iteratively performs more lookups on sources determined by intermediate results rather than looking up the entire list of relevant sources in a single pass. On the positive side, if one can live with partial results, e.g., for star-shaped queries such as (1), this approach has no need for maintaining indexes since only the correspondence between source and contained identifiers is used.

- **Schema-Level Indexes (SLI)** A second approach, mainly based on distributed query processing relies on schema-based indexes (e.g., [27], [7]). Here, the query processor keeps an index structure with properties (i.e., predicates) and/or classes (i.e., objects of `rdf:type` triples) which occur at certain sources, and uses that structure to guide query processing.

Using such schema-based indexes the incompleteness problem of direct lookups is alleviated while only using lightweight index structures. The drawback here is that instance-level descriptions are missing, i.e., (i) only queries which contain schema-level elements can be answered, and (ii) on very commonly used properties (e.g., `foaf:knows`, `foaf:name`), this index selects a (possibly too) large portion of all possible sources.

- **Data Summaries (DS)** A third approach, and the one we are advocating in this paper, uses a combined description of instance- and schema-level elements to summarise the content of data sources. We cannot keep every data item in this index, so we use a summarising index – a data summary – which represents an approximation of the whole data set. The DS approach uses more resources than the schema-level indexes, however, adds the ability to cover also query patterns including instance-level queries. Since the DS return sources which possibly contain answers to a query directly (i.e., taking joins into account), this approach may be viewed as subsuming both direct lookups and schema-level indexes. Further, a DS can be updated incrementally as the query processor obtains new/updated information about sources.

3 Source Selection using Data Summaries

The main idea of our approach is, in order to identify relevant sources, to index RDF triples provided by the sources by first transforming them into a numerical data space (applying hash functions) and then indexing the resulting data items with a data summary. In our work, we use an index structure called QTree [15] – originally developed for top-k query processing [14] – as data summary. In the following, we describe the basic principles of this structure as well as its usage for source selection.

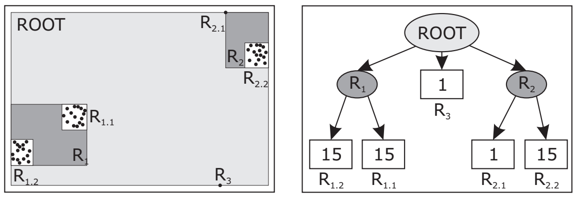
3.1 Source Indexing using the QTree

The QTree basically is a combination of histograms and R-trees [8] inheriting the benefits of both data structures: indexing multidimensional data, capturing attribute correlations, dealing with sparse data, efficient look-ups, and supporting incremental construction and maintenance. Like the R-tree, a QTree is a tree structure consisting of nodes defined by minimal bounding boxes (MBBs). MBBs of all nodes always cover all MBBs of their children and the subtrees rooted by them. Thus, an MBB describes the multidimensional region in the data space that is represented by the node the subtree underneath. Because R-trees are used to manage data items, leaf nodes in R-trees contain the data items that are contained in their MBBs. However, for our purposes we cannot hold detailed information about all data items. Rather, we have to reduce memory consumption by approximating this information.

Thus, in order to limit memory and disk consumption, we replace subtrees with special nodes called buckets. Buckets correspond to histogram buckets or bins and are always leaf nodes in the QTree – and leaf nodes are always buckets. Data items are represented by the buckets in an approximated version. As the construction of the QTree aims at grouping data items with similar hash values into the same bucket, we can use the MBBs as a good basis for approximation. As mentioned above, in our case data items are points in the multidimensional space whose coordinates are obtained by applying hash functions to the components (S, P, O) of RDF triples provided by the data sources. These components correspond to dimensions in a three-dimensional QTree.

Only buckets contain statistical information about the data items contained in their MBBs. In principle, a bucket might contain any kind of statistics, but for the purpose of this work we consider buckets capturing the number of data items contained in their MBBs (count). This means, each bucket contains the number of triples whose values (subject, predicate, object) are mapped onto coordinates that are contained in the bucket's MBB – the MBB being defined by $[S.low, S.hi], [P.low, P.hi], [O.low, O.hi]$.

The total number of buckets as well as the size of a QTree can be controlled by two parameters: (i) b_{max} denoting the maximum number of buckets in the QTree and thus limiting memory consumption, (ii) f_{max} describing the maximum fanout (i.e., the number of child nodes) for each non-leaf node. Note that the size of a QTree is independent from the number of represented data items – it only depends on these two parameters.



(a) Data and Regions (b) Hierarchy

Figure 2: Two-dimensional QTree example

Details on constructing and maintaining a QTree are beyond the scope of this paper. Thus, in the following we only sketch the basic idea and refer the interested reader to [15]. The QTree is constructed incrementally by inserting one data item after another. For each data item p, we first check whether it can be added to an existing bucket that encloses p's coordinates. In this case, the bucket statistics are updated accordingly (by incrementing the number of contained data items). Otherwise, we traverse the QTree beginning at the root node in each level looking for a node whose MBB completely encloses p. Once we have arrived at a node whose children's' MBBs do not contain p, we create a new bucket for p and insert it as a new child node.

In order to enforce the two constraints, b_{max} and f_{max}, we have to merge buckets and child nodes, respectively, if the number of buckets in the QTree or the fanout of inner nodes violates the constraints. For this purpose, we use a penalty function that represents the approximation error caused by merging two buckets and merge the pair of sibling buckets that minimises this penalty. The expensive check of all pairs is avoided by maintaining a priority queue [15].

To capture details on which RDF triples are provided by which source, we store not only the number of data items per bucket but also the ids of sources whose triples are represented by the bucket. Basically, there are two possible approaches: (i) we can simply keep a list $\mathcal{S}_B$ of source ids and a bucket cardinality c_B, or (ii) we maintain the number of triples c_B^s in each bucket B per source $s \in \mathcal{S}_B$, i.e., each bucket B contains a list of s, c_B^s pairs. For ease of explanation, in the following we stick to the first approach. In Section 3.2.2, we pick up the second approach, as it allows for a more sophisticated estimation of result cardinality or rather the number of results a source contributes to.

3.2 Source Selection

After having introduced the main concepts and characteristics of the QTree, let us now discuss how to use the information provided by this structure to decide on the relevance of sources to answer a particular query.

3.2.1 Triple Pattern Source Selection

As joins are expressed by conjunctions of multiple triple patterns and associated variables, a prerequisite for join source selection is the identification of relevant sources for a given triple pattern.

In order to determine relevant sources, we first need to identify the region in data space that contains all possible triples matching the pattern. Therefore, we need to convert a triple pattern into a set of coordinates in data space. For this purpose, we use the same hash functions that we used for index creation to obtain coordinates for a given RDF triple. However, in contrast to obtaining hash values for RDF triples provided by the sources, triple patterns of queries might contain variables. Because of these variables, in general we have to work with regions instead of points. Thus, for each literal, predicate or URI in a given triple pattern we apply the hash functions and set the minimum and maximum coordinates of the queried region to the obtained hash values. For each variable, we set the minimum and maximum coordinates to the minimum/maximum possible hash values in the respective dimension. Algorithm 1 summarises the complete procedure to determine relevant buckets (and thus, sources).

Input: BGP b, QTree QT, min/max dimensional extensions $dimSpec$
Output: list of relevant buckets

```
for $i = 0$ to $2$ do
    if $b[i] \neq variable$ then
        $R[i].low = hash(b[i])$;
        $R[i].hi = hash(b[i])$;
    else
        $R[i].low = dimSpec[i].low$;
        $R[i].hi = dimSpec[i].hi$;
    end
end
$\mathcal{B} = \emptyset$;
for $B \in QT : B$ overlaps $R$ do
    $O = B.\text{overlap}(R)$;
    $c_O = c_B \cdot \frac{\text{size}(O)}{\text{size}(B)}\}$;
    $\mathcal{B} = \mathcal{B} \cup \{(O, c_O, \mathcal{S}_B)\}$;
end
return $\mathcal{B}$
```

Algorithm 1: *identifyRelevantBuckets(BGP, QTree, dimSpec)*

After having determined the queried region R, we only need to find all buckets in the QTree that overlap R. As the QTree, similar to the R-tree, has a hierarchical structure, the lookup procedure follows similar rules: starting at the root node we need to traverse child nodes if their MBBs overlap R until we arrive at the buckets on leaf level.

After having identified all buckets with overlapping MBBs, we determine the percentage of overlap with R. Let size(R) denote the size of a region R, c_B the number of data items (cardinality) represented by bucket B and O the overlapping region of B and R. Then, the cardinality of O is calculated as $c_B \cdot \frac{\text{size}(O)}{\text{size}(B)}$. Based on this overlap, the bucket's source ids, and the cardinality (i.e., the number of represented RDF triples) we can determine the set of relevant sources and the expected number of RDF triples per source – assuming that triples are uniformly distributed within each bucket. Thus, the output of the source selection algorithm is a set of buckets, each annotated with information about the overlap with the queried region, source ids, and the associated cardinality.

3.2.2 Join Source Selection

In order to determine which sources provide relevant data for a join query, we first need to consider the triple patterns (BGPs) that a join query consists of in separate. In principle, we could return the union of all sources relevant for the individual BGPs (Section 3.2.1) as the result of the join source selection. However, it is likely that there are no join partners for data provided by some of the sources, although they match one BGP. Thus, we consider the overlaps between the sets of obtained relevant buckets for the BGPs with respect to the defined join dimensions and determine the expected result cardinality of the join.

The crucial question is how we can discard any of the sources relevant for single BGPs, i.e., identify them as irrelevant for the join. Unfortunately, if a bucket is overlapped, we cannot omit any of the according sources, because we have no information which sources contribute to which part of the bucket. In order not to miss any relevant sources, we can only assume all sources from the original bucket to be relevant. Sources can only be discarded if the whole bucket they belong to is discarded, such as the smaller bucket for the second BGP in Figure 3.

The result of a join evaluation over two BGPs is a set of three-dimensional buckets. Joining a third BGP requires a differentiation between the original dimensions, because the third BGP can be joined with any of them. This means, for instance, after a subject-subject join we have to handle two different object dimensions. To this end, a join between two three-dimensional overlapping buckets results in one six-dimensional bucket with an MBB that is equivalent to the overlap. In general, a join between n BGPs results in a $(3 \cdot n)$-dimensional join space.

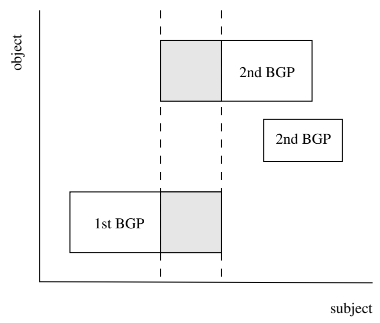


Figure 3: QTree join between first and second BGP

Figure 3 illustrates the first step of join source selection on example query (2) of the introduction, assuming that the first join is processed over the triples for subject `?X1`. For illustration purposes, we only show subject and object dimensions, as the predicate is fixed in both BGPs (i.e., the figures correspond to a slice of the actually three-dimensional space). Figure 3 exemplary illustrates a bucket that corresponds to the result of the source selection algorithm for the first BGP and shows two buckets corresponding to the second BGP. Both overlapping buckets are constrained by their overlap in the join dimension, which is the subject dimension. Other dimensions are not constrained. Thus, the shaded parts of both buckets represent the result buckets of the join.

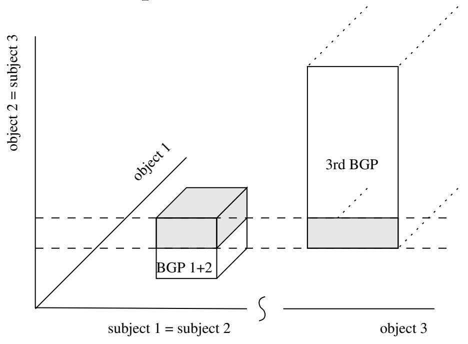


Figure 4: QTree join with third BGP

Figure 4 illustrates the next join for example query (2), assuming that it is processed on `?X2` (object-subject join between 2nd and 3rd BGP). Again, for illustration purposes, we omit the predicate dimensions and show equal dimensions on the same axis (slices of the six-dimensional space reduced to the three shown dimensions).

```
**Input**: Query $q$, QTree $QT$, min/max dimensional extensions $dimSpec$
**Output**: list of relevant sources

1  $\mathcal{J}_0 = \emptyset$;
2  forall buckets $B \in$ identifyRelevantBuckets($q$.BGP[0],$QT, dimSpec$) do
3  |   $O = B$.overlap($q$.BGP[0]);
4  |   $\mathcal{J}_0 = \mathcal{J}_0 \cup \{(O, c_B \cdot \frac{\text{size}(O)}{\text{size}(B)}, \mathcal{S}_B)\}$;
   end
6  for $i = 1$ to $|q.\text{BGP}| - 1$ do
7  |   $\mathcal{J}_i = \emptyset$;
8  |   forall buckets $L \in J_{i-1}$ do
9  |   |   forall buckets $R \in$ identifyRelevantBuckets($q$.BGP[$i$],$QT, dimSpec$) do
10 |   |   |   $d_L = q$.joindim[$i-1$]; $d_R = q$.joindim[$i$];
11 |   |   |   if $\exists O_L = L[d_L]$.overlap($R[d_R]$) then
12 |   |   |   |   $O_R = R[d_R]$.overlap($L[d_L]$);
13 |   |   |   |   $c_{O_R \times O_L} = \frac{c_L \cdot \frac{\text{size}(O_L)}{\text{size}(L)} \cdot c_R \cdot \frac{\text{size}(O_R)}{\text{size}(R)}}{\max(L[d_L].hi - L[d_L].low, R[d_R].hi - R[d_R].low)}$;
14 |   |   |   |   $\mathcal{J}_i = \mathcal{J}_i \cup \{O_L \times O_R, c_{O_R \times O_L}, \mathcal{S}_L \cup \mathcal{S}_R)\}$;
   |   |   |   end
   |   |   end
   |   end
   end
19 return $\bigcup_{B \in \mathcal{J}_{|q.\text{BGP}|-1}} \mathcal{S}_B$
```

Algorithm 2: *identifyRelevantSources(Query, QTree, dimSpec)*

Algorithm 2 sketches the whole algorithm for join source selection. In general, source selection will result in multiple buckets for each BGP. The overlap has to be determined for the cross-product of all input buckets (lines 8 and 9). We determine the buckets for each BGP separately and join them afterwards (line 9), which allows us to use existing methods for determining the overlap between the resulting buckets.

The loop in line 6 shows that we process all joins sequentially, storing the results in sets $\mathcal{J}_i$. We insert the result buckets of join i into a new $(3 \cdot (i+1))$-dimensional join space $\mathcal{J}_i$. Note that, after the first join, two of the six dimensions are equal. Handling them separately is just for ease of understanding and implementation. The $\times$ operator in line 14 symbolises the operation of combining two buckets while increasing the number of dimensions accordingly. This means, the three dimensions from O_R are added to the $3 \cdot i$ dimensions of O_L, together forming the $3 \cdot (i+1)$ dimensions of the result bucket. The new cardinality $c_{O_R \times O_L}$ (line 13) of the resulting bucket is determined using the percentage of overlap for both buckets (cf. Section 3.2.1 and line 4) and assuming uniform distribution in both buckets. The set of relevant sources $\mathcal{S}_{O_R \times O_L}$ is a union over the sets from both buckets. Finally, $\mathcal{J}_i$ serves as input for the next join (line 8).

3.3 Source Ranking

As source selection is approximative, the set of relevant sources will usually be overestimated, i.e., contain false positives (note that false negatives are impossible as we consider all QTree buckets matching any BGP of the query). Moreover, some queries may actually be answered by a large set of sources, such that a focus on the most important ones (i.e., those producing the most results) becomes important. Both issues suggest to introduce a kind of ranking for sources identified as being relevant to answer the query. There are two different general approaches that could be used to rank sources:

- *external ranking*: ranking based on an independent/externally computed notion of the sources' relevance
- *cardinality ranking*: ranking based on cardinality

External ranking may be calculated using external information (i.e., a source rank ala PageRank) or may be computed locally. Retrieving the information from, for instance, search engines, requires additional costly lookups. Local computing requires to process a lot of additional information, which again has to be retrieved by lookup queries or requires additional statistics maintained locally. An advantage of cardinality ranking is that we do not need any external knowledge. All necessary information is provided by the QTree buckets that are obtained as a result from the join source selection algorithm introduced above. The idea is to estimate the number of results $\mathcal{R}_s$ that each source $s \in \mathcal{S}$ contributes to (i.e., it holds one of the input triples). The ranks are assigned to sources according to the values of $\mathcal{R}_s$ in descending order.

As presented above, each bucket B provides an estimated cardinality c_B and a list of associated sources $\mathcal{S}_B$. In order to obtain a ranking value for a source (resembling its importance), we can just assume uniform distribution and assign $c_B/|\mathcal{S}_B|$ to each source of a bucket, while summing over all buckets. In early tests we recognised that this ranks sources very inaccurately. A simple modification of the QTree, which results in constant space overhead, is to record the cardinality c_B^s for each source of a bucket separately. More specifically, c_B^s estimates the number of results in B that source s contributes to, summed over all joined triples. Thus, $c_B = (\sum_{s \in \mathcal{S}_B} c_B^s)/jl_B$, where jl_B represents the join level of B (i.e., the number of BGPs that have been joined to form one data item in B). This helps to overcome the assumption of a uniform distribution in the bucket. The number of results a source contributes to is determined as:

$$\mathcal{R}_s = \sum_B c_B^s$$

Algorithm 2 can be adapted straightforward. Basically, we have to apply the formulas from lines 4 and 13 separately for each source, while substituting c_B by c_B^s, c_L by c_L^s and c_R by c_R^s, respectively.

This is still a rough approximation, but, as we show in Section 5, it indicates the actual importance ranking of sources in a satisfyingly accurate manner. This is grounded in probability laws, by which the probability that a source contributes to a fraction of a bucket (the region resulting from the join overlap) increases with its total number of data items in the bucket.

4 Data Summary Construction & Maintenance

In this section, we discuss possible approaches to build, expand and maintain a QTree index. We identify two main tasks, namely 1) building an initial version of a QTree and 2) expanding the index with new information of sources, referring to 1) as the *initial phase* and the *expansion phase* for 2), respectively. Once we have an initial version, we can use SPARQL queries to further explore new sources and expand the index in the expansion phase. Next, we will briefly present different approaches for each of the two phases.

4.1 Initial Phase

The initial phase is an important task with high relevance for queries and the expansion of the index. Once the QTree contains the source summaries, SPARQL queries can be evaluated against the index and the resulting relevant sources for query answering can be gathered from the Web. Users can adjust and influence the completeness of query results and the likelihood to discover new interesting sources in the expansion phase. If a user wants to have complete answers, he has to ensure that the QTree contains all relevant sources for the query. Further, the selection of seed sources can also influence the likelihood to discover new and interesting sources. On the one hand, seed sources, selected from a nearly isolated subgraph of the Linked Data Web, can lead to almost complete answers for queries. Assuming that resources are well interlinked in the Linked Data Web, it is very unlikely that other sources, not contained in the isolated subgraph, will contain relevant information to answer the particular query. On the other hand, these seed sources, isolated in the Linked Data network, will decrease the likelihood to explore other interesting Linked Data sources.

On the contrary, randomly selecting sources from the Linked Data Web as seeds increases the exploration of further sources, but decreases the completeness of answers. In general, we identified two different approaches for the initial phase and discuss them with respect to the above criteria:

- **Pre-fetching/crawling sources** The most obvious approach is to fetch seed sources for the QTree from the Web using a Web crawler. An advantage of this approach is that existing Web crawling frameworks and crawling strategies can be used to gather the seed URIs. The QTree can be adjusted wrt. to answer completeness and expansion likeliness by specifying the crawl scope. Especially, random walk strategies generally lead to representative samples of networks and thus result in seed sources that could serve as good entry points to further discover interesting sources [12]. The quality of query answers will depend on the selection of the seed sources and depth/exhaustiveness of the crawl.

- **SPARQL queries** The second approach is essentially starting with an empty QTree and using an initial SPARQL query to collect the initial sources for the QTree build. The index is expanded on further queries, cf. next subsection. Given a SPARQL query, an agent iteratively fetches the content of the URIs selected from bound variables of the query. At least, one dereferenceable URI in the SPARQL query is required as a starting point. Thus, essentially this may be viewed as starting with the plain DL approach mentioned in Section 2.

The decision which strategy to choose strongly depends on the application scenario and has to be chosen accordingly.

4.2 Expansion Phase

The second important phase is the expansion of the QTree index. Given a SPARQL query, it is very likely that the initialised QTree may contain information about dereferenceable URIs that are not (yet) indexed. In this case, the QTree should be updated with the new discovered URIs to increase the completeness of answer sets for the next time a query is executed. This expansion can be done either offline by collecting the list of new URIs and indexing them in batch processes or by integrating the content of new URIs at runtime. Further, we distinguish between pushing or pulling sources into the QTree:

- **Push of sources** is a passive approach to get new data indexed into the QTree. With passive expansion we refer to all methods that involve users or software agents to notify the QTree about new sources. This can be done by either a service similar to search engines' ping services[4] or by submitting the document directly.

- **Pull of sources** is an active approach to index new data from the Web. One way to achieve this is to perform lazy fetching during query execution. Lazy fetching refers to the process of dereferencing all new URIs needed to answer a query. This particularly fits well with an initial phase based on SPARQL queries, as outlined above. The completeness of queries and the possibility to expand the QTree with new sources depends on the initial query and can be expected to increase gradually with more queries expanding the index.

The latter sounds appealing since it solves the cold-start problem elegantly, by performing a plain DL approach on the first query and successively expanding the QTree with more relevant sources. Note that this expansion could be interleaved with prefetching one or two rounds further at each new query, thus accelerating the expansion of the QTree.

In this section, we have considered the problem of construction and maintenance on a high level – dealing mainly with the question of how to obtain information from the sources. On a lower level, we also need to consider the quality of the indexes themselves, in this case the quality of the QTree. Ideas on how to construct and maintain indexes, especially the QTree, in the presence of updates are discussed in [16], we believe that these strategies, although being developed for a different application scenario, can be adapted in a way that makes them beneficial also for the systems we focus on in this paper. Although these are important issues that have to be dealt with in general, we neglect them for the remainder of this paper and instead focus on the problem of source selection in the presence of joins.

[4]such as for instance `http://pingthesemanticweb.com/` or Sindice [22]

5 Evaluation

In this section, we present experiments performed on a fixed crawl. On the basis of a set of generated sample queries, we evaluate the performance for determining relevant sources on the QTree and the time elapsed to evaluate the query in memory. The accuracy and quality of the source selection are evaluated on the basis of a benefit measure. Most important for evaluating the practicability of the approach is to measure the impact of source ranking. Last but not least, we also simulate the DL approach and compare it to our method. As the focus of this work is on query processing, we only include basic measurements for index build time; we use the on-disk storage space requirements as a proxy for use of main memory.

We expect the QTree approach to be a lightweight but efficient and effective method to limit the search for query answers to only a subset of relevant sources. However, due to its approximate character, source selection cannot be absolutely accurate. For this we expect the introduced ranking to be a well-suited method for directing search to the most relevant sources. In comparison to the DL approach, our method should be capable of handling more types of queries in reasonable time.

5.1 Setup

Using a breadth first crawl of depth four starting at Tim Berners-Lee foaf file[5], we collected about 3 million triples from about 16.000 sources. All experiments are performed on a local copy of the so gathered data using Java 1.5 and a maximum of 3 GB main memory. 100 sample queries corresponding to two general classes were randomly generated. The first class of sample queries are star-shaped queries with one variable at the subject position. The second type of queries are path queries with join variables at subject and object positions. Figure 5 shows abstract representations of these query classes.

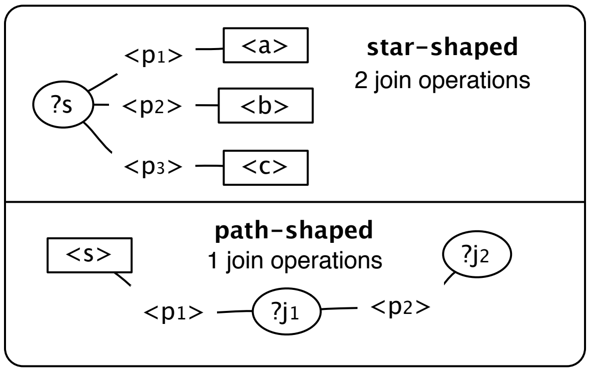


Figure 5: Abstract illustration of used query classes

The star-shaped queries were generated by randomly picking a subject from the input data and arbitrarily selecting distinct outgoing links. Then, we substituted the subject in each BGP with a variable. Path queries were generated using a random walk approach. We randomly chose a subject and performed a random walk of pre-defined depth to select object URIs. The result of such

[5]http://www.w3.org/People/Berners-Lee/card

a random walk was transformed into a path-shaped join by replacing the connecting nodes with variables. Using these approaches, we generated sample queries containing one, two and three joins operation. In the following section we use `P-`n to denote path queries with n join operations and `S-`n to denote star-shaped queries with n join operations, respectively. `BGP` refers to a queries containing only one BGP and no joins. Error bars, if shown, represent minimal and maximal values measured over all tests. The data set represents a heterogeneous and well-linked collection of documents hosted on various domains and with different numbers of RDF triples. Most of the sources are manually generated by Semantic Web affiliated users and URIs are reused among documents (e.g., DBpedia or publication/conference URIs). The query classes of choice are generally understood to be representative for real-world use cases and are used also to evaluate other RDF query systems (e.g., [21]).

5.2 Results

The measured time to insert one triple into the QTree is 4ms on average, while the final QTree requires a disk size of around 22M in serialised form. As the original data is of size 561 MB, this corresponds to a compression ratio of 96%. In the following, we present the results of four different evaluation aspects and finally discuss the results.

5.2.1 Quality of Source Selection

First, we show the quality achieved for source selection. Based on the total number of sources T in the data, the number of estimated sources E and the number of sources R actually needed to answer a query we calculate the benefit $1.0 - \frac{E}{T}$ for all queries. The benefit measures the number of sources that can be skipped in the query process, compared to the naïve approach of simply querying all known sources. In other words, the benefit gives an idea how much we safe, i.e., how many sources we can safely discard from querying. Figure 6 shows the benefit for various query types. We observe a benefit of above 80% for the star-shaped queries, while for path queries we achieve benefits of about 20%, 40% and 60%. The high benefit shows that our approach is very well suited to prune the search space of all sources. The difference between query classes is due to the fact that star queries are answered by significantly fewer sources than path queries, which usually span a large number of documents. Thus, the benefit for path queries cannot be as high as for star queries. However, the number of possibly relevant sources can still be in the number of thousands. This highlights the importance of an accurate source ranking, which is evaluated next.

5.2.2 Impact of Ranking

As mentioned before, an accurate ranking scheme is mandatory in the presence of a huge number of relevant sources. To show the impact of the ranking introduced in this work, we measured how many result triples we can determine and how many queries we can completely answer when querying only top-k ranked sources. We show according results for reasonable values of k, namely 10,50,100 and 200. Figure 7 and 8 illustrate the results of this test. In addition, Figure 9 shows the average maximal k that would be required to answer a query completely (i.e., to achieve 100% in Figure 7). The figure further shows the number of actually relevant sources. We can conclude that the introduced ranking is powerful and important for praxis. The recall values for the plots in Figure 7 are above 50% for 4 out of 7 test with the top-200 sources. Inspecting the ratio

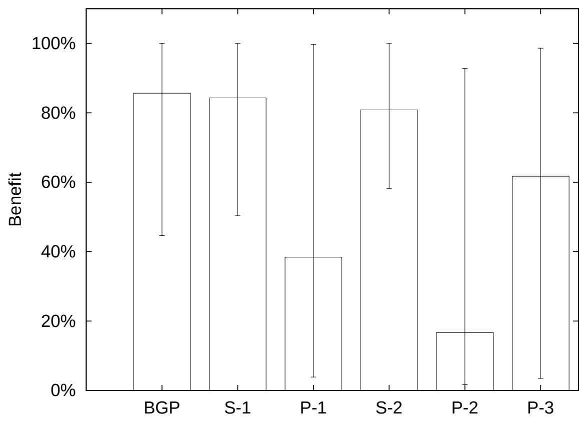


Figure 6: Benefit of source selection

of completely answered queries for the query types we observe that the path queries dominate the star-shaped queries. This is a nice complement to the higher benefit values for star-shaped queries. Figure 9 shows that the absolute error in the number of selected sources increases with the complexity of queries.

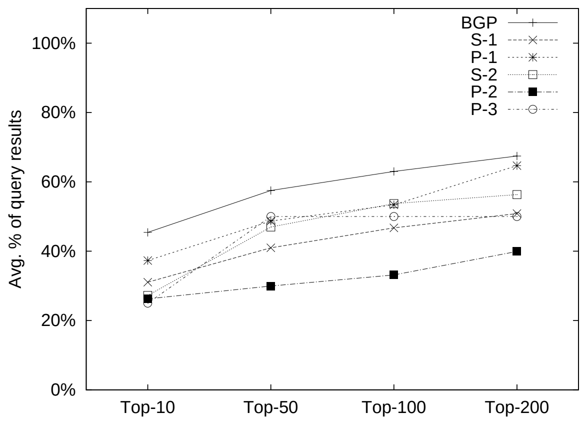


Figure 7: Impact of ranking, recall of triples

5.2.3 Query Execution Time

A crucial aspect besides the quality and benefit of the source selection is its performance, i.e., the actual time needed to answer queries. Figure 10 shows the average time required to estimate relevant sources (*qtree*) and to afterwards actually evaluating the query on the content stored in memory (*query*). The average query time for all queries is below 10 seconds, with some outliers of maximal 100 seconds. This difference in the query times results from the number of relevant sources, which is in parts significantly high (according to the QTree, but also the actual number of relevant sources for some queries). Similar times can be observed for source selection on the QTree.

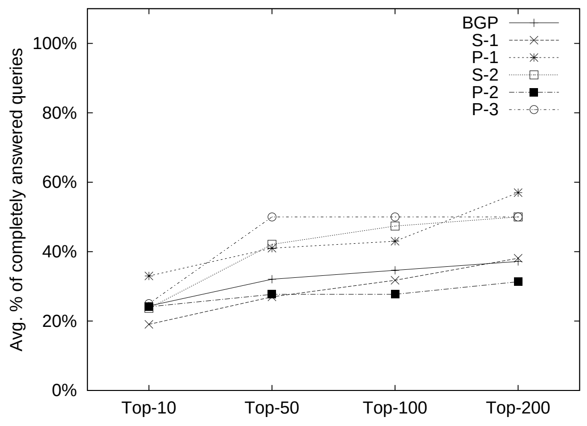


Figure 8: Impact of ranking, answer completeness

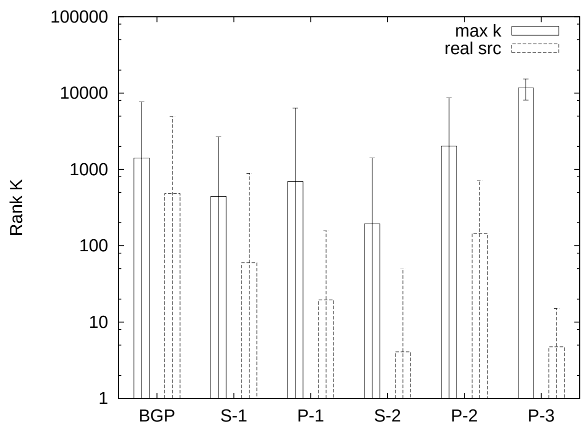


Figure 9: Impact of ranking, maximal k

The difference in here is also due to the number of buckets that have to be checked while answering single BGPs on the QTree, as query times increase with the number of buckets. The shown query times underline the applicability and practicability of our approach for a real-world application.

5.2.4 Comparison with Other Approaches

Finally, we compare our proposed solution with another possible approach, namely the DL approach. We implemented a local generalised version of the algorithm for a fair comparison with our proposed solution. Since the DL approach performs, by design, live HTTP lookups, we cannot expect the results to be completely accurate. An important difference compared to live lookups is the fact that RDF data, adhering to the Linked Data principles, have in general a relation between instance and source URIs. But, some of the instance URIs contained in our corpus may originate from data dumps instead from direct lookups. Despite this difference, an evaluation based on crawl data reflects the general limitations of the DL approach. Figure 11 shows that the DL approach is capable of returning results only for star-shaped queries with less then 2 joins.

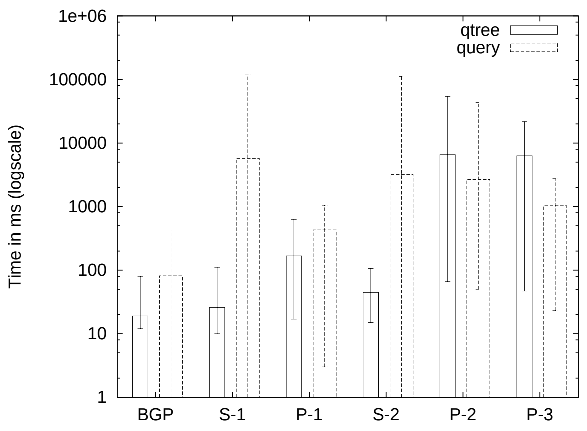


Figure 10: Query time

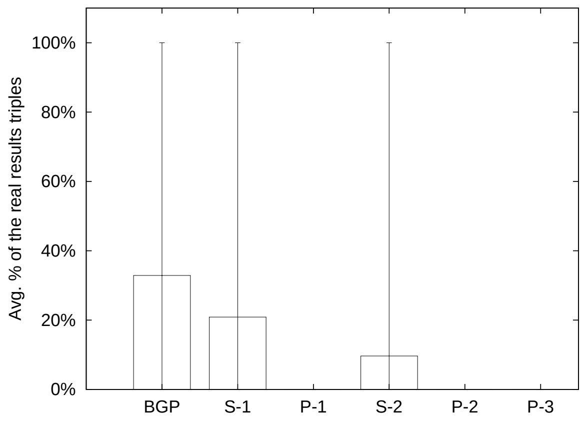


Figure 11: Applicability of the DL approach

5.3 Discussion

The evaluation shows that our novel approach is very promising and practicable for efficiently querying the Linked Data Web. The problems of state-of-the-art solutions can be eliminated successfully by the use of small index structures like the QTree. As expected, this is only practicable if an accurate ranking is applied. We were able to show that even a straightforward cardinality-based ranking is well suited to achieve this task. Our proposed solution is applicable for real-world scenarios, given the presented index and query times and the precision and impact of the top-k ranking. A client, able to perform multithreaded lookups and set up with an appropriate timeout for fetching the content of the estimated sources, can answer queries with live results in less then a minute using an index of 4% size of the original data. Almost all our expectations were met by the evaluation. Just the precision of the QTree index is slightly below our expectations and can benefit from according opimisations. Summarising, the proposed approach represents a novel, efficient and effective way of supporting source selection for live queries over the Linked Data Web. It is in a state ready for real-world applications, although the very promising results can still be tuned.

6 Related Work

An implementation of the naïve Data Lookup approach, i.e., iterative query processing with dereferencing bound URIs, has been recently presented by Hartig et al. [10]. As already sketched in Section 4, we believe our approach can be viewed as fruitfully complementing and generalising/expanding this straightforward approach towards more complete and versatile query answering over Linked Data.

Database systems have exploited the idea of capturing statistics about data for many years by using histograms [17], primarily for selectivity and cardinality estimates over local data.

The majority of work on distributed query optimisation assumes a relatively small number of endpoints with full query processing functionality rather than a possibly huge number of "dumb" sources containing small amounts of data. Stuckenschmidt et al. [27] proposed an index structure for distributed RDF repositories based on schema paths (property chains) rather than on statistical summaries of the graph-structure of the data. RDFStats [19] aims at providing statistics for RDF data that can be used for query processing and optimisation over SPARQL endpoints. Statistics include histograms, covering e.g., subjects or data types, and estimates cardinalities of selected BGPs and example queries. The Vocabulary of Interlinked Datasets (voiD)[6] is a format for encoding and publishing statistics such as basic histograms in RDF. The QTree contains more complete selectivity estimates for all BGPs of distributed Linked Data sources and the ability to estimate selectivity of joins.

A recent system using B^+-trees to index RDF data is RDF-3X [21]. To answer queries with variables in any position of an RDF triple, RDF-3X holds indexes for all possible combinations of subject, predicate and object, an idea introduced in [9]. RDF-3X uses in sophisticated join optimisation techniques based on statistics derived from the data. In contrast to our work, the approach uses a different data structure for the index and focuses on centralised RDF stores rather than distributed Linked Data sources.

Peer-to-peer systems (P2P) leverage statistical data for source selection using so-called routing indexes. Crespo et al. [5] introduced the notion of routing indexes in P2P systems as structures that, given a query, return a list of interesting neighbours (sources) based on a data structure conforming to lists of counts for keyword occurrences in documents. Based on this work, other variants of routing indexes have been proposed, e.g., based on one-dimensional histograms [23], Bloom Filters [24], bit vectors [20], or the QTree [15]. A common feature across these systems is to use a hash function to map string data on numerical data space or bits. In contrast to our work, the focus of query optimisation in P2P systems is to share load among multiple sites and optimisation has to take place at every site based on the local routing indexes.

7 Conclusion & Future Work

We have presented an approach for evaluating queries over RDF published as Linked Data, based on an index structure which summarises the content of data sources. We have shown how the index structure can be used to select relevant sources for conjunctive query answering, and how to process joins over relevant sources with an optional prioritisation via ranking. We have discussed strategies for constructing such data summaries from a static dataset or dynamically during query

[6]`http://rdfs.org/ns/void`

evaluation, and presented experimental results and discussion of our approach on synthetically generated queries over a Web crawl from 16k sources consisting of 3m RDF triples. We have shown that our approach is able to handle more expressive queries and return more complete results to queries compared to previous approaches.

While our initial results are promising, there remain several issues and future directions to explore. The possible optimisations reach from the index build and maintenance, over the integration of reasoning to an approach of a fully decentralised architecture and algorithms. An ongoing task will be to improve the current implementation with the overall goal to provide a stable and efficient library to the community. This involves low level data structure issues to improve the runtime and memory usage for join estimations and in general the maintenance of the whole QTree. In the following, we highlight some challenges and possible approaches for various open questions in more detail.

DS Build and Updates Our evaluation results show that, while the actual query processing gains a huge benefit from the DS approach, scalability is still an issue. On one hand this applies to building the DS. In the current approach, the size of the index and the time for inserting statements still depends heavily on the number of sources and triples. Preferably, we would like to have an index structure that shows constant overhead. One idea is to generate a constant-size summary approximating each source separately. Unfortunately, this cannot scale to millions of sources (even one bucket per source QTree would be too much). Moreover, querying millions of separate QTrees for each query will never scale. Ideas to overcome this issue are, among others, *(i)* to build separate QTrees and use one spanning QTree that combines all buckets and refers to the according "child QTrees" rather than actual sources, and *(ii)* to develop strategies for combining several source QTrees on demand (e.g., if they provide similar data).

The suggested cold-start build of the index is also worth some closer investigation. We plan to implement and evaluate different approaches as highlighted in Section 4, which are *(i)* the crawling of data from a list of seed URIs with different scopes like in a random walk or a focused crawl, and *(ii)* using an initial SPARQL query to fill the QTree. Both approaches result in different initial views of the Linked Data Web and thus can be used for special use cases, ranging from application-specific data mining support over a specialised and focused subset of Linked Data to a general initial index which enables discovery of other interesting and related sources.

We also explore efficient update and maintenance strategies, especially for DELETE and INSERT operations. Possible approaches are a periodical full rebuild of the QTree, which can be quite expensive, and partial rebuilds. For the latter we explore the possibility to take into account the information stored in the source id-cardinality maps together with the adaption of the bucket count to keep the number of operations and changes low. Another approach worth to investigate is the idea of dynamic QTree expansion based on user queries, as sketched in Section 4. To this end, we aim to deploy a query engine with a populated QTree for public user queries and investigate – by in parallel feeding the user queries into an empty QTree – how a QTree purely built on real user queries evolves.

Ranking and Accuracy Currently, we are exploring possibilities to increase the accuracy of the QTree summary. A crucial point in this is how to choose buckets to merge as soon as the maximal number of buckets is exceeded. At the moment, this is oriented at the usual approach for multi-dimensional data summaries, which is to minimise the area of the resulting bucket. In the setup presented here, it might be worth to also or only regard the number of sources contributing to the resulting bucket. Another approach is to regard the sources as a fourth QTree dimension. But, the resulting approximation can result in new problems, as there is no obvious strategy how to cluster sources. A specific problem is also the choice of hashing function. In our tests, different hashing functions produced in parts significantly different results. Obviously, a clustering (i.e., order-preserving) hashing function results in higher accuracy. Open questions in this context are which order to preserve, what range of hash keys, how to find a general hashing function, etc. Apparently, the choice of hashing does not only influence the accuracy, but also the performance of the whole approach.

Further, the applied ranking is a crucial point of the whole approach. Even with a straightforward cardinality ranking we achieve satisfying results. In general, there exists a tradeoff between the completeness of query results and the time required to evaluate queries. Restricting the number of lookups via cardinality ranking reduces the overall processing time in our current approach. We will investigate what types of ranking could be used to further improve the accuracy of the lookups.

Query Optimisation Our current implementation is only a proof of concept and we believe that we can improve the performance by solving and improving implementation details. We also see potential for improvement by optimising the source estimation for join queries. This is highly related to the structure of our index (e.g., one QTree for all sources vs. separate QTrees). We will evaluate these approaches case by case. This goes along with more query optimisation techniques, such as join order optimisation, for which the introduced data summaries are especially promising.

Second, it would be intuitive, given that we act directly on the Web, to also integrate other online query endpoints to increase the recall of the query results. Ideas to include these are, among others, to integrate and execute the APIs of semantic search engines, such as SWSE or Sindice, and to additionally query publicly available SPARQL endpoints. How this can be done in an efficient way and if we can gain an significant benefit needs to be evaluated in detail in the future.

Third, we are currently discussing how a caching mechanism besides the use of HTTP proxy can help us to improve the overall performance of our system and to reduce the caused network traffic. It is worth to investigate how often sources change their content and especially which type of statements in the source are deleted or added. The underlying idea is to identify statements or patterns of statements that are either rather static or dynamic. As an example, we strongly believe that type statements (e.g., instance is of type person) will not change very often over time, compared to stock quote or sensor information. Caching the former and evaluating the latter live on-demand would gain a significant performance increase and reduce necessary HTTP lookups to a minimum.

Reasoning Performing reasoning over the collected data would allow for returning consistent and extended results adhering to the specified semantics. However, future work will have to study existing and related work to identify promising approaches to integrate reasoning (to a certain extend) into the query processing and index build/maintenance.

Decentralised and Distributed Architecture Last but not least, we should highlight that QTrees are also applicable in a fully decentralised distributed query processing scenario, where peers are able to independently process and forward queries themselves. We will investigate the possibilities to which extend peers can act as stockholders for their own data and discuss their integration into query processing. This implies issues like trust and privacy, completeness of results and robustness, but also architectural questions like super-peer topologies in the presence of data sources with limited query processing capabilities. We are going to analyse the wealth of existing works in this area and to evaluate the tradeoff between benefit and complexity of such a setup in the context of ad-hoc queries on the Linked Web of Data.

References

[1] T. Berners-Lee. Linked data, July 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[2] D. Brickley, L. Miller. FOAF Vocabulary Spec. 0.91, 2007. `http://xmlns.com/foaf/spec/`.

[3] G. Cheng, Y. Qu. Searching linked objects with falcons: Approach, implementation and evaluation. *JSWIS*, 5(3):49–70, 2009.

[4] K. G. Clark, L. Feigenbaum, E. Torres. SPARQL protocol for RDF, Jan. 2008. W3C Rec., `http://www.w3.org/TR/rdf-sparql-protocol/`.

[5] A. Crespo, H. Garcia-Molina. Routing indices for peer-to-peer systems. *ICDCS '02*, p.23–32, 2002.

[6] M. d'Aquin, C. Baldassarre, L. Gridinoc, S. Angeletou, M. Sabou, E. Motta. Characterizing knowledge on the semantic web with watson. *EON'07*, p.1–10, 2007.

[7] R. Goldman, J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. *VLDB'97*, p.436–445, 1997.

[8] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD '84*, p.47–57, 1984.

[9] A. Harth, S. Decker. Optimized index structures for querying RDF from the web. *3rd Latin American Web Congress*, p.71–80, 2005.

[10] O. Hartig, C. Bizer, J.-C. Freytag. Executing sparql queries over the web of linked data. *ISWC'09*, 2009.

[11] D. Heimbigner, D. McLeod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278, 1985.

[12] M. R. Henzinger, A. Heydon, M. Mitzenmacher, M. Najork. Measuring index quality using random walks on the web. *Computer Networks*, 31(11-16):1291–1303, 1999.

[13] A. Hogan, A. Harth, J. Umbrich, S. Decker. Towards a scalable search and query engine for the web. *WWW'07*, p.1301–1302, 2007.

[14] K. Hose, M. Karnstedt, A. Koch, K. Sattler, D. Zinn. Processing Rank-Aware Queries in P2P Systems. *DBISP2P'05*, p.238–249, 2005.

[15] K. Hose, D. Klan, K. Sattler. Distributed Data Summaries for Approximate Query Processing in PDMS. *IDEAS '06*, p.37–44, 2006.

[16] K. Hose, C. Lemke, K. Sattler. Maintenance Strategies for Routing Indexes. *Distributed and Parallel Databases*, 26(2-3):231–259, 2009.

[17] Y. Ioannidis. The History of Histograms (abridged). *VLDB '03*, p.19–30, 2003.

[18] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, Dec. 2000.

[19] A. Langegger, W. Wöß. RDFstats - an extensible RDF statistics generator and library. *8th Int'l Workshop on Web Semantics, DEXA*, 2009.

[20] M. Marzolla, M. Mordacchini, S. Orlando. Tree Vector Indexes: Efficient Range Queries for Dynamic Content on Peer-to-Peer Networks. *PDP'06*, p.457–464, 2006.

[21] T. Neumann, G. Weikum. RDF-3X: a RISC-style Engine for RDF. *VLDB Endow.*, 1(1):647–659, 2008.

[22] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello. Sindice.com: A document-oriented lookup index for open linked data. *JMSO*, 3(1), 2008.

[23] Y. Petrakis, G. Koloniari, E. Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer Systems. *DBISP2P '04*, p.16–30, 2004.

[24] Y. Petrakis and E. Pitoura. On Constructing Small Worlds in Unstructured Peer-to-Peer Systems. *EDBT Workshops*, p.415–424, 2004.

[25] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF, Jan. 2008. W3C Rec., `http://www.w3.org/TR/rdf-sparql-query/`.

[26] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. *ESWC'08*, p.524–538, Tenerife, Spain, 2008.

[27] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. *WWW'04*, p.631–639, 2004.