World Wide Web (2011) 14:495–544
DOI 10.1007/s11280-010-0107-z

# Comparing data summaries for processing live queries over Linked Data

**Jürgen Umbrich · Katja Hose · Marcel Karnstedt · Andreas Harth · Axel Polleres**

Received: 15 May 2010 / Revised: 30 October 2010 /
Accepted: 21 December 2010 / Published online: 7 January 2011


**Abstract** A growing amount of Linked Data—graph-structured data accessible at sources distributed across the Web—enables advanced data integration and decision-making applications. Typical systems operating on Linked Data collect (crawl) and pre-process (index) large amounts of data, and evaluate queries against a centralised repository. Given that crawling and indexing are time-consuming operations, the data in the centralised index may be out of date at query execution time. An ideal query answering system for querying Linked Data live should return current answers in a reasonable amount of time, even on corpora as large as the Web. In such a live query system source selection—determining which sources contribute answers to a query—is a crucial step. In this article we propose to use lightweight data summaries for determining relevant sources during query evaluation. We compare several data structures and hash functions with respect to their suitability for building such summaries, stressing benefits for queries that contain joins and require ranking of results and sources. We elaborate on join variants, join ordering and ranking. We analyse the different approaches theoretically and provide results of an extensive experimental evaluation.

**Keywords** index structures · Linked Data · RDF querying

J. Umbrich · M. Karnstedt · A. Polleres
Digital Enterprise Research Institute, National University of Ireland, Galway, Ireland

K. Hose
Max-Planck-Institut für Informatik, Saarbrücken, Germany

A. Harth (✉)
Institute AIFB, Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: harth@kit.edu

## 1 Introduction

The recent developments around Linked Data have led to the exposure of large amounts of data on the Semantic Web amenable to automated processing in software programs [6]. Linked Data sources use RDF (the Resource Description Framework) in various serialisation syntaxes for encoding graph-structured data, and the Hypertext Transfer Protocol (HTTP) for data transfer. 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 that form a global data graph. Advanced data integration and decision-making applications require query processing over the global graph. We can distinguish two broad directions:

- data warehousing or *materialisation-based approaches* (*MAT*), which collect the data from all known sources in advance and pre-process, index and store the combined data in a central database; queries are evaluated using the local database.
- *distributed query processing approaches* (*DQP*), which parse, normalise and split the query into sub-queries, determine the sources containing results for sub-queries, and evaluate the sub-queries against the sources directly.

Most current approaches enabling query processing over RDF data, such as Semantic Web search engines [11, 16, 32, 51], are based on *MAT*. Centralised approaches provide excellent query response times due to extensive preprocessing carried out during the load and indexing steps, but suffer 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 query, MAT involves unnecessary data gathering, processing, and storage since large portions of the data might not be used for answering the particular query. Lastly, due to replicated data storage, data providers have to give up sole sovereignty on their data (e.g., they cannot restrict or log access anymore since queries are answered against a copy of the data).

On the other end of the spectrum, *DQP* approaches typically assume processing power attainable at the sources themselves, which could be leveraged in parallel for query processing. Such distributed or federated approaches [28] 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 site that issues the query. The potential drawback, however, is that DQP systems cannot give strict guarantees on query performance since the integration system relies on a large number of potentially unreliable sources.

DQP is a well-studied problem in the database community [42], however, existing *DQP* approaches do not scale above a relatively small number of sources and heavily exploit schema information typically not available on linked RDF data. Previous results for query processing over distributed RDF stores [57] assume, similar to approaches from the traditional database works, few endpoints with local query processing capabilities and sizable amounts of data, rather than many small Web resources only accessible via HTTP GET.

Currently only a few data sources on the Web provide full query processing capabilities (e.g., by implementing SPARQL [12, 56], a query language and protocol

for RDF), but we still would like to eschew the cost of maintaining a full index of the data at a central location. To this end, in the present work we investigate the middle ground between the two extremes of MAT and DQP and propose the use of *histogram-based data summaries*[1] concisely describing the contents of—as typical for current Linked Data—a large number of sources accessible via HTTP GET, which, in contrast to classic data integration scenarios, are of small size in the range of a few kilobytes to megabytes. These data summaries will form the basis for (i) *source selection* by ruling out sources that cannot contribute to the overall query or joins within (sub-)queries, and (ii) *query optimisation* to provide the optimal join order and determine the most important sources via ranking.

Source selection is a crucial step that affects the efficiency of the execution of queries. The goal is to identify the data sources that possibly provide results for the given query or, in other words, to eliminate sources from the query plan that do not contribute to the result. In classic distributed databases source selection is supported by (query or view) expressions describing the fragmentation of a global table or schema.

Our aim is to narrow the gap between the two extreme approaches (MAT vs. DQP) for processing queries over Linked Data sources directly. We introduce and compare several index regimes that support a local query processor in selecting sources—which are accessed during query runtime— that contribute answers to a query. Note that our approach is applicable in certain scenarios where (i) direct lookup on sources during query time is a desirable feature in order to guarantee current results and (ii) keeping a full local index is too expensive; MAT systems which fully index datasets (e.g., [24, 50, 65]) may be better suited in scenarios where fast lookups are more important than freshness of results and keeping a full local index is affordable.

In a preliminary version of this work we have investigated the use of an approximate multidimensional indexing structure (a QTree [35]) as a data summary [25]. In the present paper we extend this study with further details on index construction, maintenance, and lookup complexity for a variety of alternative indexing approaches. Specifically, our new contributions compared to [25] are as follows:

- We present and discuss a general model for querying Linked Data and evaluate the complexity of the query processing with regard to an extended set of indexing approaches.
- We provide comprehensive details about algorithms for index creation, index lookups and join processing, with a theoretical analysis of space and time complexity.
- We present a comprehensive experimental evaluation of different data summaries (including hashing methods) based on synthetic queries over Linked Data obtained from the Web.

In the remainder of this paper we introduce in Section 2 basic concepts around Linked Data and outline several methods for answering queries over Linked Data,

[1] In the following we will simply speak of "data summaries" when we refer to such histogram-based data summaries.

including those using data summaries. In Section 3, we detail two possible approaches for data summarisation and source selection based on multidimensional histograms and QTrees, including approaches for constructing and maintaining these data summaries. In Section 4 we discuss different alternatives for hashing, i.e., the translation from identifiers to a numerical representation that both data summaries rely upon. We present algorithms for source selection, join processing, and discuss source ranking in Section 5. The experimental setup for evaluation is described in Section 6 followed by a detailed discussion of the experimental results comparing different data summaries and hashing alternatives in Section 7. In Section 8, we place our approach in the context of related work and conclude with a summary and an outlook to future work in Section 9.

## 2 Querying Linked Data

In this section we introduce basic concepts (RDF, Linked Data) and present a generic query processing model which can be instantiated with specific mechanisms for data source selection.

### 2.1 RDF and Linked Data

The Resource Description Framework [27, 45] defines a data format for publishing schema-less data on the Web in the form of (*subject*, *predicate*, *object*) triples. These triples are composed of unique identifiers (URI references), literals (e.g., strings or other data values), and local identifiers called blank nodes as follows:

**Definition 1** (RDF triple, RDF term, RDF graph) Given a set of URI references $\mathcal{U}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, a triple (s, p, o) $\in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is called an *RDF triple*, We call elements of $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ *RDF terms*. Sets of RDF triples are called *RDF graphs*.

The notion of graph stems from the fact that RDF triples may be viewed as labelled edges connecting subjects and objects. RDF published on the Web according to the following principles is called *Linked Data* [4]:

1. URIs are used as names for things: in contrast to the HTML Web where URIs are used to denote content (documents, images), on the Semantic Web URIs can denote entities such as people or cities.
2. URIs should be dereferenceable using the Hypertext Transfer Protocol (HTTP): a user agent should be able to perform HTTP GET operations on the URI.
3. Useful content in RDF should be provided at these URIs: a Web server should return data encoded in one of the various RDF serialisations.
4. Links to other URIs should be provided for discovery: a user agent should be able to navigate from an entity to associated entities by following links, thus enabling decentralised discovery of new data.

In short, these principles shall ensure that RDF graphs published on the Web contain again HTTP-dereferenceable URIs returning more information about the respective URIs upon lookup. Linked Data assumes a correspondence (in URI

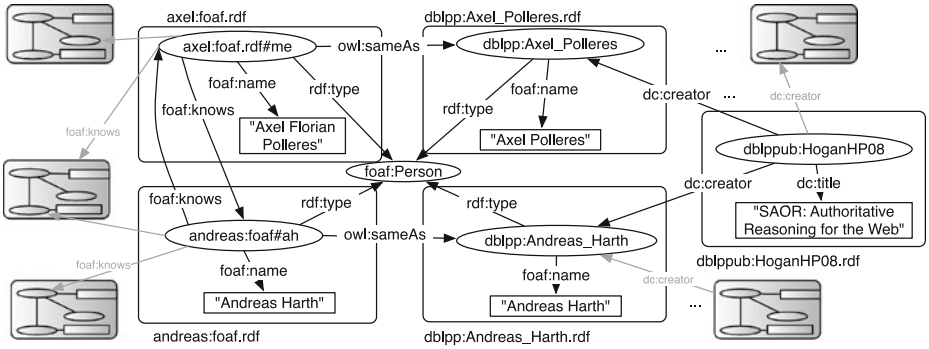



**Figure 1** Linked Data in RDF about persons and their publications.

syntax or via HTTP redirects) between a URI of a resource and the data source. For an example of a syntactic correspondence consider the resource URI http://harth.org/andreas/foaf#ah which is described at the source URI http://harth.org/andreas/foaf. HTTP redirects work as follows: a user agent (a Web browser or other piece of software) performs a lookup on a URI, and the Web server answers with a new location for the URI. For example, a lookup on http://xmlns.com/foaf/0.1/knows redirects to a HTML file http://xmlns.com/foaf/spec/index.html when dereferenced via a Web browser, or to an RDF/XML file http://xmlns.com/foaf/spec/index.rdf when accessed via an RDF-aware client. User agents specify their preferred content format via HTTP's `Accept` header for content negotiation.

Maintaining a distinction between the identifier (denoting an entity) and the data source (the location of data about the entity) is important for the ability of describing both the entity and the data source separately. A correspondence between the entity URI and the source URI—either via syntactic means or via HTTP redirects—allows for decentralised lookups without introducing a global catalogue associating entity URI with data source URI.

There are various serialisation formats for publishing RDF on the Web, normative RDF/XML or RDF embedded in HTML (RDFa). We will sometimes visually represent RDF as graphs (e.g. in Figure 1) or use the more readable N3 [5] notation in this paper, where RDF triples are written as whitespace separated triples and graphs are written as '.'-separated lists of such triples. Brackets (<>) denote URIs and quotes ("") denote literals. Blank node identifiers start with '_:'. Namespaces can be introduced to abbreviate full URIs as *namespaceprefix:localname*, e.g., a parser expands foaf:name in combination with the syntactic definition of the namespace prefix to http://xmlns.com/foaf/0.1/name.

N3 syntax is also used in the SPARQL [56] query language for RDF which we use in examples. Conjunctive queries in SPARQL[2] consist of so-called *basic graph patterns*.

[2]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 layered on top of conjunctive query functionality.

**Definition 2** (Basic graph pattern, triple pattern) A *triple pattern* is an RDF triple that may contain variables (prefixed with '?') instead of RDF terms in any position. A set of triple patterns is called *basic graph pattern (BGP)*.

Moreover—analogous to the common notion of acyclic conjunctive queries—we denote *acyclic BGPs* as those where the following graph is acyclic: as nodes we take all the triple patterns in the BGP with the triple patterns that share a variable (join variables) being connected by an edge.

### 2.2 Example

As an example, consider a scenario in which sources publish interlinked data about people, the relations between them and their publications, as depicted in Figure 1. The figure shows five RDF graphs covering data about Andreas and Axel as currently available on the Web: RDF from their personal homepages,[3] which cover personal information described using the Friend-of-a-Friend (FOAF) vocabulary [7], two graphs incorporating information about them at DBLP,[4] and one of their joint publications at DBLP[5] using the Dublin Core vocabulary.[6]

The following simple SPARQL query asks for names of Andreas' acquaintances (variable `?n`) and consists of two triple patterns joined by variable `?f`:

```
SELECT ?n WHERE {
 <http://harth.org/andreas/foaf#ah> foaf:knows ?f.    (1)
 ?f foaf:name ?n. }
```

The next query asks for people who know authors of the article identified via URI dblppub:HoganHP08. Such a query could be used to rule out possibly conflicting reviewers:

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

Both queries are acyclic.

### 2.3 Issues with accessing Linked Data

Accessing Linked Data shares properties of traditional (HTML) Web crawling, given that Linked Data uses HTTP and URIs as underlying technologies. Web crawlers perform lookups on a vast number of sources operated by a diverse set of owners. Thus, Web crawlers have to act politely, i.e., be careful not to overload servers

[3] http://harth.org/andreas/foaf.rdf, http://www.polleres.net/foaf.rdf

[4] http://dblp.l3s.de/d2r/resource/authors/Andreas_Harth, http://dblp.l3s.de/d2r/resource/authors/Axel_Polleres

[5] http://dblp.l3s.de/d2r/page/publications/conf/aswc/HoganHP08

[6] http://dublincore.org/2010/10/11/dcterms.rdf

with requests. Since typical Web crawlers are multi-threaded and perform HTTP lookups in parallel, care has to be taken to schedule URI lookups in a way that does not impact the operation of sources. In addition, server administrators can use a specification based on `robots.txt` files[7] to allow or disallow clients access to certain URIs on a server, i.e. using `robots.txt` files a server administrator can guide the behaviour of crawlers that adhere to the protocol. For our further considerations, we assume that a dedicated access component takes care of content negotiation and RDF extraction, and adheres to basic politeness rules common to Web crawlers. In other words, the access component performing lookups on URIs simply returns an RDF graph (which can be empty, e.g. in case of broken links).

### 2.4 Generic query processing model

Typically, distributed query processing (DQP) involves the following steps for transforming a high-level query into an efficient query execution plan:

1. parse query into internal representation;
2. normalise query by application of equivalence rules;
3. un-nest and simplify the logical query plan;
4. optimise by re-ordering query operators;
5. select sources that contribute (partial) answers;
6. generate a physical query plan (replace the logical query operators with specific algorithms and access methods);
7. execute the physical query plan.

For a complete discussion of query processing please see [18]. We focus of the main problems of processing such queries which are (i) finding the right sources to contain possible answers that can contribute to the overall query and (ii) efficient parallel fetching of content from these sources. Query processing in our approach works as follows:

– prime a compact index structure with a seed data set (various mechanisms for creating and maintaining the index are covered in Section 3);
– use the data-summary index to determine which sources contribute partial results for a conjunctive SPARQL query $Q$ and optionally rank these sources;
– fetch the content of the sources into memory (optionally using only the top-k sources according to auxiliary data for ranking stored in the data summary);
– perform join processing locally, i.e., as opposed to full DQP approaches we do not assume that remote sources provide functionality for computing joins.

The strategy is a reasonable compromise under the following assumptions, which we believe to hold for a wide range of Linked Data sources and typical queries:

1. The overall data distribution characteristics within different sources do not change dramatically over time, and can be captured in a data summary in lieu of a full local data-index.
2. Source selection and ranking can reduce the amount of fetched sources to a reasonable size so that content can be fetched and processed locally.

[7] http://www.robotstxt.org/

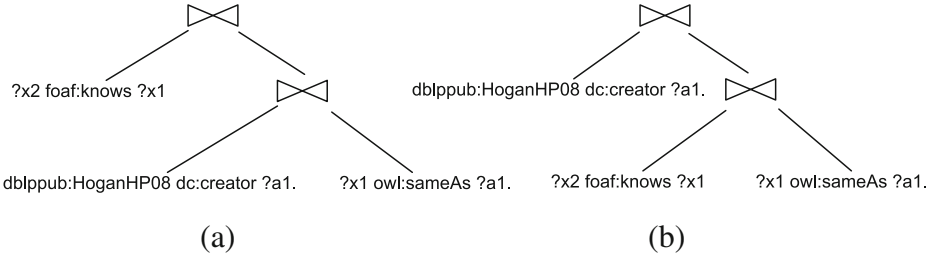



**Figure 2** Join trees for query (2).

In the following, we focus on the part of the query plans covering acyclic conjunctive BGPs in SPARQL queries, which can be encoded as a join tree. Figure 2 depicts two possible join trees for query (2).

Analogous to (equi-)joins in standard relational algebra we view a join as a binary operator taking either a triple pattern or the result of another join as input. Each join results in a table of values bound to variables, combining bindings to equal values on both sides of the join per row. For ease of exposition, in this section we assume that variable bindings from the left operator are substituted in the right operator, and the right triple pattern is then evaluated to return bindings (nested-loop join). We get back to alternative join approaches such as hash joins and particularly the importance of the hashing functions for our own approach in Sections 4 and 5. A potentially large number of sources can contribute bindings to each of the triple patterns, so the query evaluator requires a method that allows for the selection of sources. This *source-selection* method takes as input a triple pattern and returns a list of data sources that potentially contribute bindings. Since accessing too many sources over the Web is potentially very costly, the choice of the source-selection method is fundamental, and strongly depends on the various possible query approaches and their underlying index structures, which we discuss next.

### 2.5 Source-selection approaches

In the following we first introduce approaches for source selection, starting with the approach that introduces the least complexity, and then describe each approach in more detail. We do not cover standard RDF indexing approaches (i.e., complete indexes [24, 50, 65]) as our premise is to perform query processing over data sources directly: rather than maintaining complete indexes and deriving bindings from index lookups, we aim at using data structures that just support the selection of data sources from which data will be retrieved and then processed locally. That is, the general idea is to use index structures to guide the query processor, while being reasonable both wrt. completeness of the answers and network bandwidth consumption. Possible approaches to evaluate queries over Web sources and particularly addressing the problem of source selection are:

– **Direct lookup (DL)**: this approach exploits the correspondence between resource URIs mentioned in the query and source URI. That is, only URIs mentioned in the query or URIs from partial results fetched during query execution are looked up directly. Since the source URIs can be derived from the URIs

**Table 1** Triple patterns supported by the different approaches.

| Approach | Triple patterns |
|---|---|
| DL | `(#s ?p ?o)`, `(#s #p ?o)`, `(#s #p #o)` and possibly `(?s ?p #o)`, `(?s #p #o)` |
| SLI | `(?s #p ?o)`, `(?s rdf:type #o)` |
| II, MDH, QT | All |

mentioned in the query, the approach does not need any index structures, but—given the structure of Linked Data—will likely only return answers for selected triple patterns (see Table 1).

- **Schema-level Indexing (SLI)** relies on schema-based indexes known from query processing in distributed databases [21, 63]. The query processor keeps an index structure of the schema, i.e, which properties (URIs used as predicates in triples) and/or classes (i.e., objects of `rdf:type` triples) occur at certain sources and uses that index to guide query processing. Triple patterns with variables in predicate position cannot be answered (see Table 1).
- **Inverted URI Indexing (II)** indexes all URIs occurring in a given data source. An II covers occurrences of URIs in sources. The II allows the query processor to identify all sources which contain a given URI and thus potentially contribute bindings for a triple pattern containing that URI. Using an inverted URI index, a query processor can obtain bindings from sources which contain the pertinent URI but for which the resource/data source correspondence as specified in the Linked Data principles does not hold. For example, source `axel:foaf.rdf` might contain additional descriptions about `andreas:foaf#ah` not mentioned in `andreas:foaf.rdf`. The index stores such mentions of URIs outside their implicitly associated source.
- **Multidimensional histograms (MDH)** use a combined description of instance- and schema-level elements to summarise the content of data sources using histograms. MDHs represent an approximation of the whole data set to reduce the amount of data stored. We will present one type of MDH in more detail in Section 3.
- **QTree (QT)**: The QTree is another approach that uses a combined description of instance- and schema-level elements to summarise the content of data sources. In contrast to the MDH where regions are of fixed size, the QTree is a tree-based data structure where regions of variable size more accurately cover the content of sources.

When we look at all possible triple patterns in the BGPs of SPARQL queries, we realise that different source selection mechanisms only cover a subset of those. At an abstract level, triple patterns can have one of the following eight forms (where ? denotes variables and # denotes constants):

```
(?s ?p ?o)  (#s ?p ?o)  (?s #p ?o)  (?s ?p #o)
(#s #p ?o)  (#s ?p #o)  (?s #p #o)  (#s #p #o)
```

Table 1 lists the triple patterns that can be answered using the respective source-selection mechanism.

Which source-selection approach to use depends on the application scenario. DL works without additional index structures, but fails to incorporate URI usage outside the directly associated source (via syntactic means—the URIs containing a #—or via HTTP redirects). Further, DL follows an inherently sequential approach of fetching relevant sources (as new sources are discovered during query execution), whereas the other mentioned indexing approaches enable a parallel fetch of relevant sources throughout. II can leverage optimised inverted index structures known from Information Retrieval, or can use already established aggregators (such as search engines) to supply sources. While not supporting full joins, II has been extended to support simple, star-shaped joins [17]. SLI works reliably on queries with specified values at the predicate position, and can conveniently be extended to indexing arbitrary paths of properties [63]. However, both II and SLI, are "exact" indexes which have the same worst-case complexity as a full index, i.e., potentially grow proportionally to the number and size of data sources (or, more specifically with the number URIs mentioned therein).[8]

In the present work, we will particularly focus on the deployment of approximate data summaries which can be further compressed depending on available space—at the cost of a potentially more coarse-grained source selection results—namely MDH and QT, which only necessarily grow with the number of data sources, but not necessarily with the number of different URIs mentioned therein. MDH is an approach inexpensive to build and maintain but may provide a too coarse-grained index which negatively affects the benefit of using the index. QT is more accurate, as the data structure is—as we will see—able to represent dependencies between terms in single RDF triples and combinations of triples, which can be leveraged during join processing, however, at increased cost for index construction and maintenance. Note that approaches such as II and SLI do not model those dependencies, which can result in suboptimal performance.[9]

We provide detailed experimental evaluation for each of the mentioned index structures in Section 7.

## 3 Data summaries

Data summaries allow the query processor to decide on the relevance of a source with respect to a given query. Querying only relevant instead of all available sources can reduce the cost of query execution dramatically.

The data summaries we focus on describe the data provided by the sources in much more detail than schema-level indexes; thus in general the number of queried sources is less for the data summary approach. Data summaries represent the data provided by the sources in an aggregated form. As summarising numerical data is

[8] While this might be viewed as a theoretical limitation not relevant in practical data, we will also see other advantages of alternative data summaries that we focus on in this paper.

[9] While the mentioned extensions of SLI [63] or extensions of II [17] partially address this issue as well, they do not cover arbitrary acyclic queries, but only fixed paths in the case of [63] or star-shaped queries in the case of [17].

more efficient than summarising strings, the first step in building a summary index is to transform the RDF triples provided by the sources into numerical space. We apply hash functions to the individual components ($S$, $P$, $O$) of RDF triples so that we obtain a triple of numerical numbers for each RDF triple. All these "numerical" triples are then used to insert the sources they originate from into a data summary, i.e., this data summary stores lists of sources for numerical ranges. Given a query, the first step is to determine relevant numerical triples that answer the query. By looking up these triples in the data summary, we obtain a set of sources potentially providing relevant data.

In the remainder of this section, we first introduce the two variants of data summaries we focus on: multidimensional histograms [38] and QTrees [35, 36, 66]. Afterwards, we discuss time and space complexity for index insertion and lookup and finally give details on how to initialise and expand data summaries in general. We discuss the influence and importance of hash functions separately in Section 4.

### 3.1 Multidimensional histograms

Histograms are one of the basic building blocks for estimating selectivity and cardinality in database systems. Throughout the years, many different variations have been developed and designed for special applications [38]. However, the basic principles are common to all of them. First, the numerical space is partitioned into regions, each defining a so-called *bucket*. A bucket contains statistical information about the data items contained in the respective region. If we need to represent not only single values but instead pairs, triples or $n$-tuples in general, we need to use multidimensional histograms with $n$ dimensions. In $n$-dimensional data space, the regions represented by the buckets correspond to $n$-dimensional hypercubes. For simplicity, however, we refer to them simply as regions. For RDF data, we need three dimensions—for subject, predicate, and object.

Data items, or triples respectively, are inserted one after another and aggregated into regions. Aggregation and thus space reduction is achieved by keeping, for each three-dimensional region, statistics instead of the full details of the data items. A straightforward option is to maintain a number of data items per region and a list of sources contributing to this count. However, we show that source selection and ranking perform better if we maintain a set of pairs (*count*,*source*)—denoting that *count* data items provided by the source *source* are represented by the region.

The main difference between histogram variations is how the bucket boundaries are determined. There is a trade-off between construction/maintenance costs and approximation quality. Approximation quality is determined by the size of the region and the distribution of represented data items—on big region for only a few data items has a higher approximation error than several small regions for the same data. The quality of a histogram also depends on the inserted data. We decided to use equi-width histograms as an example representative for MDH, because they are easy to explain, apply to a wide range of scenarios and can be built efficiently even if the exact data distribution is not known in advance.

For this kind of histograms, given the minimum and maximum values of the numerical dimensions to index and the maximum number of buckets per dimension, each dimensional range is divided into equi-distant partitions. Each partition defines the boundaries of a region/bucket in the dimension. The upper part of Figure 3 shows

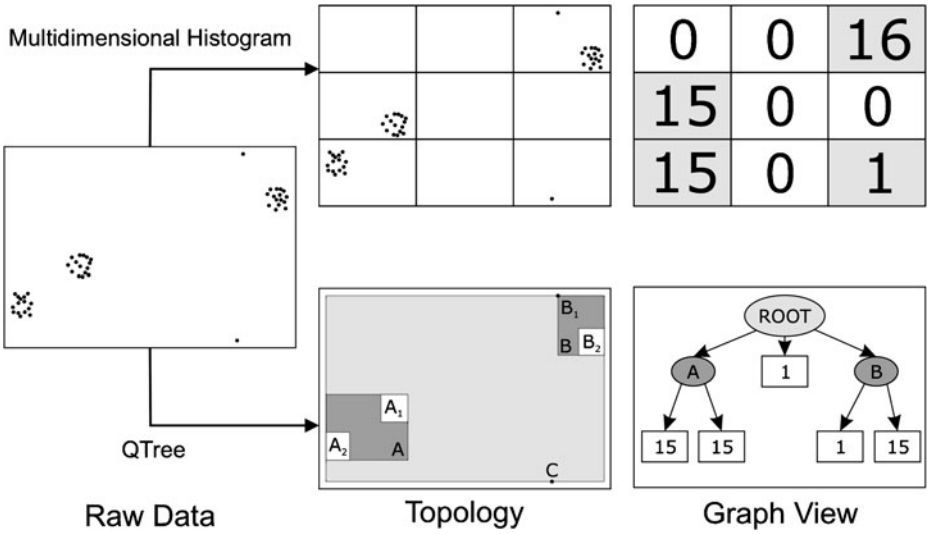



**Figure 3** Example of a data summary. The *left* column shows the coordinates corresponding to hash values for the data items to insert. The *middle* column shows the bucket regions, and the *right* column shows the buckets with the assigned statistical data and, in case of the QTree, the hierarchy between inner nodes and buckets.

a two-dimensional example of a multidimensional equi-width histogram with the number of buckets per dimension set to three.

Given an RDF triple, a lookup entails computing the corresponding numerical triple by applying the same hash function to the RDF triple that has been used for constructing the histogram and retrieving the bucket responsible for the obtained numerical triple. The bucket contains information about which sources provide how many data items in the bucket's region. Hence, we only need to consider the found relevant sources. However, there is no guarantee that the sources actually provide the RDF triple that we were originally looking for (false positives). The reason is that a bucket does not represent exact coordinates in data space but a region which also covers coordinates for RDF triples not provided by any indexed source.

### *3.1.1 Space complexity*

Determining space consumption for the histogram is straightforward; denoting the total number of buckets used by a multidimensional histogram as $b_{\max}$, the number of sources by $c_{\text{sources}}$, and considering the maximum number of (*count*, *source*) pairs per bucket, we can state that it requires $O(b_{\max} \cdot c_{\text{sources}})$ space (Table 2).

**Table 2** Space complexity of QTrees and multidimensional histograms.

| Space complexity | |
|---|---|
| Multidimensional histogram | $O(b_{\max} \cdot c_{\text{sources}})$ |
| QTree | $O(b_{\max} \cdot c_{\text{sources}})$ |

### *3.1.2 Runtime complexity*

For the multidimensional histogram introduced above, determining the bucket that a numerical triple $d$ has to be inserted into has complexity $O(1)$—having arranged the buckets in an array, the coordinates of the searched bucket can easily be determined based on the static boundaries of the regions. The insertion itself can be done in $O(\log\ c_{\text{sources}})$ by finding the bucket's (*count*, *source*) pair corresponding to $d$'s source and adapting the *count* value—using a Java TreeMap to manage the pairs. A lookup resembles the procedure of determining the bucket responsible for the data item to insert and is therefore also done in $O(\log c_{\text{sources}})$ (Table 3).

## 3.2 QTree

The QTree—originally developed for top-$k$ query processing [35, 36, 66]—is a combination of multidimensional histograms and R-trees [23] and therefore inherits benefits from both data structures: indexing multidimensional data, capturing attribute correlations, efficiently dealing with sparse data, allowing efficient look-ups, and supporting incremental construction and maintenance.

In contrast to the histograms introduced above and similar to R-trees, QTrees are hierarchical structures. They consist of nodes representing regions in the data space. The region of a node always covers all the regions of its child nodes. Data items are only represented by leaf nodes—in analogy to the multidimensional histogram introduced above we refer to leaf nodes as buckets and store the same information in them. The lower part of Figure 3 shows an example QTree with all regions of inner nodes and buckets as well as the hierarchy between them.

In contrast to standard histograms, QTrees do not necessarily cover all the data space but only regions containing data. Thus, in case of sparse data the histograms introduced above use same sized regions for areas representing many data items as well as for areas containing only a few data items. QTrees, however, use regions of variable sizes covering only areas containing data. The number of nodes in a QTree is determined by two parameters: $b_{\max}$ denoting the maximum number of buckets in the QTree and $f_{\max}$ describing the maximum fanout (i.e., the number of child nodes) for each non-leaf node.

Although R-trees and QTrees share the same principle of indexing data by organising multidimensional regions hierarchically, they differ in a substantial way: whereas the QTree approximates the indexed data to reduce space consumption,

**Table 3** Time complexity of QTrees and multidimensional histograms.

| Time complexity | |
|---|---|
| Multidimensional histogram insert | $O(\log c_{\text{sources}})$ |
| Multidimensional histogram lookup | $O(\log c_{\text{sources}})$ |
| QTree insert | $O(b_{\max} \cdot f_{\max}^2 + c_{\text{sources}} \cdot \log\ c_{\text{sources}})$ or $O(\log b_{\max} \cdot f_{\max}^2 + c_{\text{sources}} \cdot \log c_{\text{sources}})$ |
| QTree lookup | $O(b_{\max} + \log c_{\text{sources}})$ or $O(\log b_{\max} + \log c_{\text{sources}})$ |

R-trees keep detailed information about inserted data items (tuples in our case). A QTree's leaf node provides statistical information about the tuples it represents, i.e., the number and origin (source) of tuples located in the multidimensional region—and no information about the tuples' coordinates. In contrast, an R-tree's leaf node keeps the exact coordinates of the tuples and therefore consumes more space than a QTree. Thus, space consumption for QTrees (maximum number of buckets determines the degree of approximation) increases only with the number of data sources whereas space consumption for an R-trees variant holding the same information increases both with the number of sources and the number of tuples.

When additional tuples are inserted into an R-tree, having access to the exact coordinates of all represented tuples allows for efficient re-balancing so that we can guarantee that the R-tree is always balanced, i.e., all leaf nodes are on the same level. As a consequence of approximation, we cannot guarantee balance for the QTree; as we do not know the exact coordinates of tuples represented by a particular region, we cannot simply split the region for re-balancing without losing accuracy/correctness. For splitting we would have to "guess" (due to the approximation) which of the resulting regions the original data should be assigned to. Therefore, instead of re-balancing algorithms like the R-tree, the QTree applies heuristics to keep itself balanced. Although balancing works well in practice, we cannot give any guarantees because it is theoretically possible to construct a QTree that conforms to a linear list [34, 66].

QTrees are constructed by inserting one data item after another. However, due to the hierarchical structure, construction is more complex and adheres to the following steps (more details [34, 66]):

(1) **Try to insert *d* into an existing bucket.** For each data item $d$, we first check whether $d$ can be added to an existing bucket whose region encloses $d$'s coordinates. If found, the bucket statistics are updated by adapting the (*count*, *source*) pair corresponding to the source that $d$ originates from. If a pair corresponding to the source already exists, we increase its *count* value by one. If such a pair does not exist, we add a new pair (1,*source*).

(2) **Find most responsible inner node and insert *d* as a new bucket.** If $d$ could not be inserted into an existing bucket, we traverse the QTree beginning at the root node. We look for a child node $p$ whose region completely encloses $d$ and proceed recursively with $p$. We stop when we cannot find such a child node. Instead, we create a bucket as a new child node with a single (1, *source*) pair representing $d$.

(3) **Enforce $f_{max}$ constraint.** The previous step might have violated the $f_{\max}$ constraint by creating a new bucket and attaching it as a child to an inner node $p$. In order to reduce $p$'s fanout, we determine the pair of $p$'s child nodes whose merging would result in a node with the smallest region. Either we create a new child node $n$ of $p$ and attach the pair of chosen siblings as child nodes to $n$, or if the bucket inserted in the previous step is part of the pair of chosen siblings, we assign the node as a new child of its sibling node. The latter case might result in the need to recursively enforce the $f_{\max}$ constraint. In addition, we apply some heuristics to prevent the QTree from degenerating into a linear list by trying to destroy inner nodes whose children could be attached to the parent node without violating the constraints.

(4) **Enforce $b_{\max}$ constraint.** If step (3) resulted in a situation where the maximum number of buckets $b_{\max}$ is exceeded, we need to reduce the number of buckets in the QTree. Hence, we search for the pair of sibling buckets whose merging would result in a bucket with the smallest region. The so-found pair of buckets is merged into a new bucket whose region minimally encloses those of the original buckets and whose set of (*count*, *value*) pairs originates from merging pairs of the original buckets—pairs referring to the same source are merged by summing up their *count* values. Since the parent of the merged siblings now has less child nodes, we try to remove the parent by attempting to assign its child nodes to its parent.

In order to avoid comparing all sibling buckets of all levels to each other each time we need to enforce the constraints, we maintain a priority queue containing at most one entry for each inner node stating its pair of child buckets to be merged next. The entry corresponding to an inner node only needs to be updated when its child buckets are added or merged.

A lookup for an RDF triple in the QTree is very similar to the lookup in multidimensional histograms. The only difference is finding a bucket that contains the numerical triple corresponding to the given RDF triple. In contrast to the histogram approach introduced above, buckets in a QTree might overlap so that we find multiple buckets for one given RDF triple. As we do not know into which one the triple has been inserted when constructing the index, we need to find all such buckets. We find these buckets by traversing the QTree starting at the root node and recursively following all paths rooted by children whose regions contain the coordinates defined by the numerical triple we are looking for—as regions are allowed to overlap, we might need to traverse multiple paths. Traversing a path ends at a bucket or when there are no further child nodes containing the coordinates. Just as for histograms false positives are possible, i.e., the index indicates the relevance of a source although in fact it does not provide the data item we were looking for.

### *3.2.1 Space complexity*

The QTree's space complexity comprises the space consumption of its main components: leaf nodes, inner nodes, and the priority queue. Note that the size of a QTree depends solely on its parameters ($f_{\max}$ and $b_{\max}$) as well as the number of sources $c_{\text{sources}}$, but is independent of the number of represented data items.

By enforcing the $b_{\max}$ constraint, we can ensure that a QTree contains at most $b_{\max}$ leaf nodes. Only leaf nodes hold statistical information about the sources in the form of (*count*, *source*) pairs. The size of each pair is fixed and the number of pairs depends on the number of sources $c_{\text{sources}}$. Thus, leaf nodes require $O(b_{\max} \cdot c_{\text{sources}})$ space.

Construction ensures that an inner node has at most $f_{\max}$ and at least two child nodes. A QTree is a tree structure where each inner node has exactly two children and thus a QTree has at most $b_{\max} - 1$ inner nodes. Allowing inner nodes to have more children, i.e., $f_{\max}$ which is always greater than or equal to 2, never increases but only reduces the number of inner nodes. Thus, a QTree has at most $b_{\max} - 1$ inner nodes. As the priority queue has at most one entry for each inner node, it cannot have more than $b_{\max}$ entries. Consequently, a QTree requires $O(b_{\max})$ space for inner

nodes and the priority queue. Hence, in total a QTree requires $O(b_{\max} \cdot c_{\text{sources}})$ for its main components altogether (Table 2).

The space complexity of both, multidimensional histograms and QTrees, highlights an earlier-mentioned advantage of these hash-based data summaries in comparison to other indexing approaches discussed in Section 2.5: the other indexes sketched there grow in the worst case with the number of indexed triples. In contrast, due to the supported adaptive approximation, histograms and QTree grow only with the number of data sources, independent from the number of indexed triples. Moreover, the total size of the data summary is adjustable by setting an appropriate $b_{\max}$ value.

### *3.2.2 Runtime complexity*

To determine runtime complexity for constructing the QTree, we need to determine the costs for each main step of the insertion algorithm—we omit proofs and refer interested readers to [66] for more details. When inserting a data item $d$, these costs are:

(1) Try to insert $d$ into an existing bucket

$$O(b_{\max} + \log\ \mathit{sources})$$

(2) Find most responsible inner node and insert $d$ as a new bucket

$$O(b_{\max})$$

(3) Enforce $f_{\max}$ constraint

$$O\left(b_{\max} \cdot f_{\max}^2\right)$$

(4) Enforce $b_{\max}$ constraint

$$O\left(f_{\max}^2 + \log\ b_{\max} + c_{\text{sources}} \cdot \log\ c_{\text{sources}}\right)$$

In the first step, we try to insert $d$ into an existing bucket. In the worst case, we might have to traverse all buckets, thus $O(b_{\max})$ time. Updating the (*count*, *source*) pairs requires $O(\log\ \mathit{sources})$ time using a Java TreeMap. Thus, time complexity is $O(b_{\max} + \log\ \mathit{sources})$ in total.

In the second step, we traverse the tree until we find a node at which we can insert $d$ as a new bucket. In the worst case, we have to visit all nodes and check if $d$ is contained in the nodes' regions: $O(b_{\max})$. The insertion of $d$ as a new bucket takes constant time $O(1)$.

The third step is more expensive as we need to enforce the $f_{\max}$ constraint possibly recursively through the tree. In the worst case, we need to make adaptations to each level in the tree, i.e., enforcement is required at most $b_{\max}$ times. For each call we need to find that pair of child nodes whose merging would result in the smallest region. Thus, we need to compare each pair of child nodes, i.e., $O(f_{\max}^2)$. Rearranging child nodes and creating a new inner node takes $O(f_{\max}^2)$ time. Thus, in total the third step requires $O(b_{\max} \cdot f_{\max}^2)$ time.

In the fourth step, we need to enforce the $b_{\max}$ constraint and reduce the number of buckets. Finding the best pair of buckets is done in $O(1)$ because all we have to do is to remove the first entry from the priority queue. Merging two buckets requires $O(c_{\text{sources}} \cdot \log\ c_{\text{sources}})$ using the Java TreeMap to organise (*count*, *source*) pairs. The priority queue needs to be updated with respect to the parent node of the merged buckets—finding the best pair of buckets takes $O(f_{\max}^2)$ and updating the priority queue takes $O(\log\ b_{\max})$ operations using a heap-based priority queue. In any case, whether the parent node can be dropped—$O(f_{\max}^2)$—or not, only one entry of the priority queue needs to be updated. Thus, in total step four takes $O(f_{\max}^2 + \log\ b_{\max} + c_{\text{sources}} \cdot \log\ c_{\text{sources}})$ time.

Performing a lookup operation in the QTree requires at most $O(b_{\max} + \log\ c_{\text{sources}})$ because in the worst case we need to visit all nodes, and lookups in the (*count*, *value*) pairs cost at most $O(\log\ c_{\text{sources}})$.

The above runtime complexities are based on the worst case corresponding to an unbalanced QTree with a tree height of $b_{\max}$. The construction algorithm cannot guarantee that we obtain a balanced tree structure, because it is theoretically possible to construct a QTree that conforms to a list-like structure. For example, if tuples are inserted sorted in a way that the next tuple's coordinates are always higher than those of the previous tuple, then we would add all the data to only one branch of the QTree. However, such an order is unlikely for real data sets and the heuristics we apply keep the QTree almost balanced in practice. Assuming that we have an almost balanced tree, insertion takes only $O(\log\ b_{\max} \cdot f_{\max}^2 + c_{\text{sources}} \cdot \log\ c_{\text{sources}})$ time and a lookup $O(\log\ b_{\max} + \log\ c_{\text{sources}})$ (Table 3).

## 3.3 Construction and maintenance

So far we have only considered one aspect of constructing data summaries, i.e., how to insert data. We have not yet considered when and what data we need to index. In this respect, we identify two main tasks: (i) creating an initial version of a data summary (*initial phase*) and (ii) expanding the summary with additional or new information (*expansion phase*).

Construction and maintenance of the data summaries are not the focus of the present work. However, we give a brief overview of possibilities and sketch the general directions which could be investigated further.

### *3.3.1 Initial phase*

Once we have constructed an initial version of a data summary, we can use it to determine a set of relevant sources for a given SPARQL query and retrieve relevant documents from the Web. The selection of seed sources to construct the initial version has a strong influence on the ability to discover new and interesting sources in the expansion phase.

Let us assume the case that our data summary covers a subgraph containing only few incoming or outgoing links to the rest of the global Linked Data Web. The lack of links to new sources decreases the probability of further extending the index. Selecting seed sources which provide many links to other documents increases the chance of discovering new sources. The selection of those well interlinked sources can be done via sampling on a random walk over the Linked Data graph or choosing the top ranked sources of existing datasets.

In general, we identify two different approaches for constructing an initial version:

(i) **Pre-fetching** The most obvious approach is to fetch an initial set of sources to index from the Web using a Web crawler. An advantage of this approach is that existing Web crawling systems can be used to gather the seed URIs. Random walk strategies, in particular, generally lead to representative samples of networks and thus result in a set of sources that could serve as a good starting point to further discover interesting sources [31]. The quality of query answers depends on the selection of the selected sources and depth/exhaustiveness of the crawl.

(ii) **SPARQL queries** Another approach is to use SPARQL queries and collect the sources to index from the answer to the queries. Given a SPARQL query, an agent iteratively fetches the content of the URIs selected from bound variables of the query (the DL approach). At least one dereferenceable URI in the SPARQL query is required as a starting point.

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

### *3.3.2 Expansion phase*

After having created an initial version of a data summary, there might still be sources whose data have not yet been indexed. Given a SPARQL query, it is very likely that the initial summary contains information about dereferenceable URIs that are not indexed. In this case, the summary should be updated with these newly discovered URIs to increase the completeness of answer sets for future queries. In this context, we distinguish between pushing and pulling sources:

(i) **Push of sources** refers to methods involving users or software agents to trigger expansion, which can be done for example via a service similar to the ping services[10] of search engines.

(ii) **Pull of sources** does not need any external triggers and can be implemented using lazy fetching during query execution. Lazy fetching refers to the process of dereferencing all new URIs needed to answer a query. The approach is similar to constructing the initial data summary using SPARQL queries.

The latter approach sounds appealing since it elegantly solves the cold-start problem by performing a plain Direct Lookup approach on the first query and successively

[10]Such as for instance http://pingthesemanticweb.com/, the sitemap submission protocol used by search engines http://www.sitemaps.org/protocol.php#submit_engine, or Sindiceś [51] ping service http://sindice.com/developers/pingApi which also supports the submission of Semantic sitemaps [15].

expanding the QTree with more relevant sources. Note that the expansion could be combined with pre-fetching for each new query, thus accelerating the expansion of the summary.

## 4 Hashing

In the following we explain how our system uses hash functions to map RDF statements to numerical values. The numerical values are inserted and stored in the buckets of the data summaries. Similar or correlated data items should be clustered and represented in the same bucket. Data summaries adhering to these criteria are ideal for query optimisation and source selection and therefore have a positive influence on precision.

In the case of multidimensional histograms, the hash function should equally distribute the input data among the buckets. The equal distribution is not a requirement for the QTree, since the buckets are adjusted to the input values.

There is a wide variety of hash functions which map strings to integer or float values. A trivial class of the hash functions interpret the encoding of a string as an integer or long value. Another widely used group of hash functions represents the string values with its computed checksum, fingerprint or digests, e.g., one can use the CRC checksum or encryption algorithms like SHA-1. More advanced hash functions try to minimise possible bijective mappings from different strings to the same hash value. Other functions are order preserving, that means that the order of the hash values reflect the order of the input data; e.g., the alphabetical order of the strings.

A common method allowing for efficient aggregation is to normalise the hash values by scaling them from the numerical range of the hash function into a smaller range. One possible way of scaling is to use a linear transformation as depicted in Figure 4. The figure also illustrates how a range scale improves clustering and leads to an uniform distribution of the data.

To define the numerical range of a data summary, we have to consider two special cases:

(1) **Target range is too big (sparse data)** If the target region is too big, most of the target range is likely not to be occupied at all. This strongly affects the quality

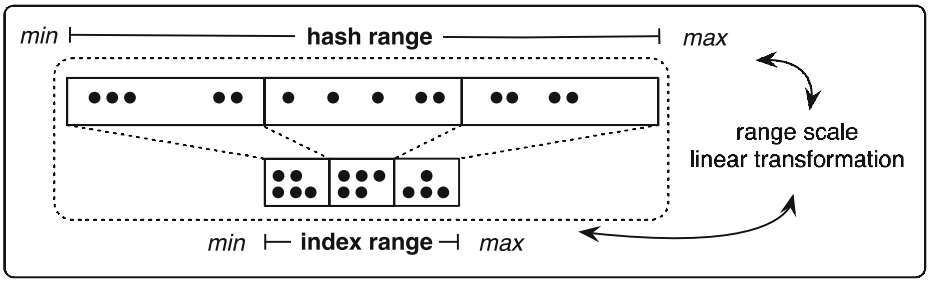



**Figure 4** Range scale.

of the multidimensional equi-width histogram whereas the QTree was designed to handle sparse data—see Section 3 for more details.

(2) **Target range is too small** If the target range is too small, we have to deal with hash value collisions, i.e., different strings are mapped onto the same numerical value although their original hash values were different. This will lead to false positive decisions for source selection.

Thus, selecting the target range is a crucial task and directly influences the query processing.

### 4.1 Hash functions

The general approach is to apply the hash function to the string value of each node in the RDF triple. Alternatively, one can consider applying different hash functions to the different types of RDF terms, namely resources, blank nodes and literals and/or considering the position of the RDF term (subject, predicate or object).

In this article, we focus on the following two hashing approaches:

*String hashing (STR)* This group of hash functions computes hash values based on checksums and fingerprints. The advantage is that these algorithms aim at providing a unique hash value for each string value and thus try to use the all the available numerical space.

*Prefix hashing (PRE)* Prefix hash functions use a prefix tree to map a string value to the value of the longest matching prefix in the tree and thus provide a good clustering for similar strings. The basic structure of a prefix tree is depicted in Figure 5. In the example we can see that all string values starting with http://xmlns.com/ are mapped to values between 2 and 5. For example, the URI http://xmlns.com/foaf/spec/name is hashed to 3. A string only consisting of the prefix http://xmlns.com/ is hashed to 5. The advantage of prefix hashing is that it provides a better clustering of similar RDF values compared to the string hashing. However, prefix hashing reduces the number of possible values, especially if the prefix tree does not contain many prefixes. In

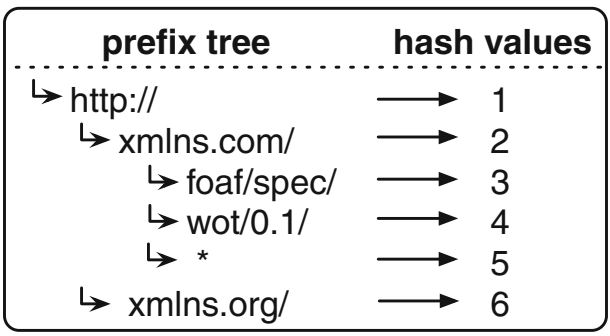


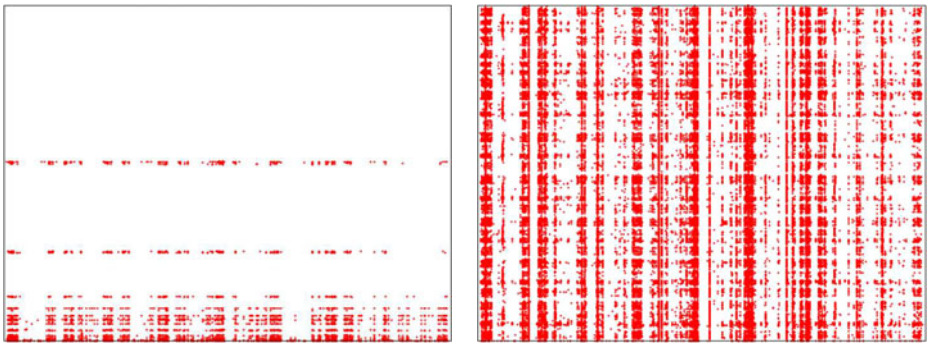

**Figure 5** Prefix hashing.

addition, we have to maintain a prefix tree which can consume a lot of space because of the number of prefixes. However, early experiments showed that a QTree with prefix hashing performs better than a string hashing in terms of the quality of the source selection.

*Mixed hashing (MIX)* The mixed hashing function combines the approaches of prefix and string hashing. Subject and object values are hashed using prefix hashing and predicate values are hashed with string hashing using checksums. The reason for applying different hash functions for the different position of the RDF terms is that earlier experiments revealed that the number of distinct predicates on the Web is rather small (in the number of ten thousand) compared to the possible number of subject and object values. A prefix hash function would map several predicates to the same hash value and we would loose important information. If we apply a string hashing function for the RDF terms at the predicate position we conserve more information because each predicate will be mapped to a unique hash value. This is especially important for the join processing as we will discuss in detail in Section 5.

### 4.2 Comparison

The distribution of the input data for mixed and string hashing is shown in Figure 6 as two-dimensional plots. We omit the prefix hashing from the figure because the distribution patterns are very similar to the mixed hashing, with the only difference that the predicate dimension contains more data points for the mixed hashing. The plots show the distribution of the hashed RDF statements for the RDF terms at the predicate (x-axis) and object (y-axis) position. We selected these two dimensions because they show best the difference between the two hashing approaches and are representative for the other dimensions. The input dataset is a breadth-first Web crawl of RDF/XML documents starting from a well connected source (more information about the dataset in Section 6). We can see that the string hashing



(a) Mixed hashing (b) String hashing

**Figure 6** 2D plots of the hash values of different hashing functions for RDF terms at the predicate (x-axis) and object (y-axis) position.

equally distributes the input data over the allocated numerical space. The mixed hashing shows a higher clustering of the input data and leaves large areas of the numerical space empty. Based on these patterns, we can conclude that in theory string hashing should be more suitable for histograms and a prefix or mixed hashing favours the QTree. Our evaluation provide several proofs that the theory holds in practice.

## 5 Source selection

An advantage of the data summaries we advocate here is that they support estimation of relevant sources by processing parts of a query plan before actually fetching any sources. In this section, we provide details on how source estimation works for single triple patterns and joins of triple patterns. Table 4 summarises the notation we use throughout the section. Note that for ease of exposition we refer to buckets that contain a single *count* value and a list of sources contributing to this value—we indicate the required modifications for buckets containing a set of (*count*, *source*) pairs where appropriate (cf. Section 3).

### 5.1 Triple pattern source selection

Single triple patterns define the leaf operators of query plans, where relevant data is extracted from the Linked Data sources. For determining the sources that can contribute to a join, we first determine the sources that can contribute to these basic triple patterns. With the help of the data summaries, source selection is achieved by determining the buckets (i.e., the data regions) that correspond to a triple pattern. Therefore, a triple pattern is converted into a set of coordinates in numerical space by applying the used hash functions to the elements of the pattern. Triple patterns containing only constants map to a single point in the three-dimensional space, while variables result in spanning the whole range of hash values for the

**Table 4** Used symbols.

| Symbol | Explanation |
|---|---|
| $\mathcal{B}, \mathcal{R}, \mathcal{L}$ | Sets of buckets (regions) |
| $B, R, L$ | Buckets (regions) |
| $\oplus$ | Bucket join operation (used in *region joins*) |
| $\mathcal{J}$ | Join space (special set of buckets) |
| $R[i]$ | $i$-th dimension of bucket $R$ |
| $R[i].hi$, $R[i].low$ | Max and min value of $R$ in $i$-th dimension |
| $c_R$ | Cardinality of bucket $R$ |
| $\mathcal{S}_R$ | Set of sources contributing to bucket $R$ |
| $s_r$ | Number of results that source $s$ contributes to |
| $\lvert\mathcal{R}\rvert$ | Number of buckets in set $\mathcal{R}$ |
| $\overleftrightarrow{R}^j$ | Extension in join dimension $j$ of bucket $R$ |
| $\overline{\mathcal{R}}^j$ | Average extension in join dimension $j$ of all buckets in $\mathcal{R}$ |
| $\frown$ | "Overlapping" relation |
| $\sqcap_j$ | Overlapping intervals in dimension(s) $j$ of two buckets |

respective dimension, thus constructing a cubic region corresponding to the triple pattern. Intuitively, several filter expressions can be included in the construction of such a *query region*. This includes all filter statements that can be mapped directly to according hash values (e.g., range expressions, but not contains expressions). Algorithm 1 summaries the complete procedure to determine relevant sources for a given triple pattern.

**Algorithm 1** Source selection for triple patterns.

**Input**: BGP $b$, QTree $QT$, min/max dimensional extensions $dimSpec$
**Output**: list of relevant buckets (containing sources)

```
1  for i = 0 to 2 do
2      if b[i] ≠ variable then
3          R[i].low = hash(b[i]);
4          R[i].hi = hash(b[i]);
       else
6          R[i].low = dimSpec[i].low;
7          R[i].hi = dimSpec[i].hi;
       end
   end
10 B = ∅;
11 for B ∈ QT : B overlaps R do
12     O = B.overlap(R);
13     c_O = c_B · size(O)/size(B)};
14     B = B ∪ {(O, c_O, S_B)};
   end
16 return B
```

Based on the constructed query region $R$, we can determine all buckets contained in the data summary that overlap with $R$. In multidimensional histograms, all buckets are inspected in sequence. In contrast, the hierarchical structure of a QTree supports to start at the root node and then to traverse all child nodes if their minimal bounding boxes (MBBs) overlap $R$. All buckets on leaf level visited by this tree traversal constitute the set of relevant buckets.

After having identified all relevant buckets, 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 the overlap, the bucket's source URIs, 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 URIs, and the associated cardinality.

### 5.2 Join source selection

The presented source selection for triple patterns (and filter statements) already reduces the number of sources that have to be fetched for processing a query. However, we can reduce that number even further if we include the join operators into the pre-processing of a query. The buckets determined for single triple patterns act as input for the join source selection. As it is likely that there are no join partners for data provided by some of the sources relevant for a triple pattern, this will reduce the number of relevant sources. Thus, we consider the overlaps between the sets of obtained relevant buckets for the triple patterns with respect to the defined join dimensions and determine the expected result cardinality of the join. In the general case, a join between two triple patterns is defined by equality in one of the dimensions. Thus, we have to determine the overlap between buckets in the join dimensions, while leaving other dimensions unconstrained.

The performance of processing joins on the data summaries depends on several factors, where the most relevant are:

1. the order of joins;
2. the actual processing of the join operation.

As in relational databases, the first point should be handled using a cost estimation for different join orders and the second one by choosing between different join implementations. Before we discuss these basic optimisations, we will illustrate the general principle of such *region joins*.

#### *5.2.1 Region joins*

The crucial question is how we can discard any of the sources relevant for single triple patterns, i.e., identify sources as irrelevant for the join. Unfortunately, if a bucket is overlapped, we cannot omit any of the contributing sources, because we have no information on which sources contribute to which part of the bucket. To not miss any relevant sources, we can only assume all sources from the original bucket to be relevant. Sources can only be discarded if the entire bucket they belong to is discarded, such as the smaller bucket $R_2$ for the second triple pattern in Figure 7. Thus, data summaries and hashing functions that result in small sets of small buckets promise to be particularly beneficial for the join source selection and the overall performance of our query processing approach.

The result of a join evaluation over two triple patterns is a set of three-dimensional buckets. Joining a third triple pattern requires a differentiation between the original dimensions, because the third triple pattern can be joined with any of them. For instance, after a subject-subject join we have to handle two different object dimensions; 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 $n$ triple patterns results in a $(3 \cdot n)$-dimensional *join space*.

Figure 7 illustrates the first step of join source selection on the basis of query (2) from Section 2. We assume that the first join is processed over the first and second triple pattern, i.e., an object-object join over `?a1` (corresponding to the join tree shown in Figure 2a). The sets of input regions for each triple pattern are determined

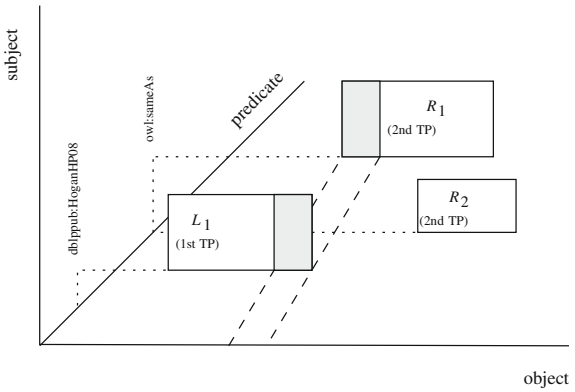



**Figure 7** Region join between first and second triple pattern.

as described in Section 5.1 on the basis of the queried predicate. For simplicity, we assume this results in only one bucket for the first and two buckets for the second triple pattern. Each resulting bucket corresponds to a slice of the three-dimensional space. With regard to the join dimension, there are two overlapping buckets. Both overlapping buckets $L_1$ and $R_1$ are constrained by their overlap in the join dimension. Other dimensions are not constrained. Thus, the shaded parts of both buckets represent the result buckets of the join. For a join between different dimensions, e.g., a subject-object join, the approach is the same. The subject dimension of the first triple pattern restricts the object dimension of the second, and vice versa.

Figure 8 illustrates how the second join from example query (2) is processed. The join involves `?x1`, i.e., an object-subject join between the second and third triple pattern. 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). The left-most bucket $L_1 \bowtie R_1$ corresponds to the bucket resulting from the first join. One bucket $S_1$ shown from the result of the third triple pattern overlaps with it, i.e., there is an overlap between the `subject 2` dimension

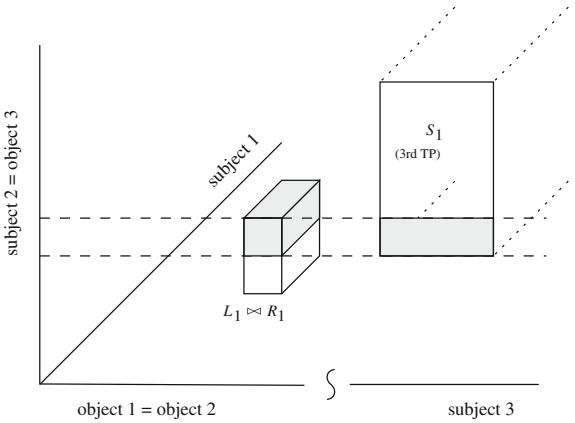



**Figure 8** Region join with third triple pattern.

of $L_1 \bowtie R_1$ (originally, the subject from $R_1$) and the object dimension of $S_1$. The resulting overlap defines the nine-dimensional result bucket, containing information about all resources that might contribute to this bucket.

### *5.2.2 Region join implementations*

The general principle of determining sources relevant for joined triple patterns can be implemented in several different ways. Basically, the different alternatives known from relational databases can be mapped to the processing of region joins. We discuss the alternatives available in our query engine in the following.

*Nested-loop join* A straightforward implementation of a region join is a *nested-loop join*. This is the implementation we first proposed in [25]. Algorithm 2 provides a detailed illustration of this implementation. The overlap between the both input sets is determined by two nested loops (lines 2 and 3). Overlapping buckets are joined in the inner loop (line 3) using existing methods for determining the overlap between buckets.

**Algorithm 2** Nested-loop join.

**Input**: join space $\mathcal{J}_{in}$ (left-hand side), set of buckets $\mathcal{R}$ (right-hand side), left/right join dimension $l/r$
**Output**: new join space (containing relevant buckets and sources)

```
1  J_out = ∅;
2  forall buckets L ∈ J_in do
3      forall buckets R ∈ R do
4          if ∃O_L = L[l].overlap(R[r]) then
5              O_R = R[r].overlap(L[l]);
6              c_{O_R⊕O_L} = (c_L · size(O_L)/size(L) · c_R · size(O_R)/size(R)) / max(L[l].hi − L[l].low, R[r].hi − R[r].low);
7              J_out = J_out ∪ {O_L ⊕ O_R, c_{O_R⊕O_L}, S_L ∪ S_R};
           end
       end
   end
11 return J_out
```

The $(3 \cdot (i+1))$-dimensional regions resulting from the $i$-th join are stored in a join space $\mathcal{J}$. This join space acts as one input for the next join. Thus, the join operator actually processes one such join space and the three-dimensional regions for a triple pattern—for the first join, $\mathcal{J}$ corresponds to the three-dimensional buckets resulting from the left-most triple pattern in the join tree. 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 $\oplus$ operator in line 7 symbolises the operation of combining two buckets while increasing the number of dimensions accordingly: 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 $c_{O_R \oplus O_L}$ (line 6) of the resulting bucket is determined using the percentage of overlap

for both buckets (cf. Section 5.1), assuming uniform distribution in both buckets. We provide details in Section 5.3. The set of relevant sources $\mathcal{S}_{O_R \oplus O_L}$ is a union over the sets from both buckets.

**Algorithm 3** Principle of nested-loop join.

**Input**: left input $\mathcal{L}$, right input $\mathcal{R}$
**Output**: join space containing overlapping buckets

```
forall L ∈ ℒ do
    forall R ∈ ℛ do
        𝒥.add(determineOverlap(L, R));
    end
end
return 𝒥
```

A simplified description of the nested-loop join principle is shown in Algorithm 3. Note that we omit the restriction to the actual join dimensions. We use this abbreviated form to show the differences to other implementations. The resulting number of operations is $\Theta(|\mathcal{L}| \cdot |\mathcal{R}|)$, i.e., we have to call method `determineOverlap` exactly $|\mathcal{L}| \cdot |\mathcal{R}|$ times.

*Index join* Intuitively, the efficiency of the join processing can be increased using special join indexes. One option is to use an *inverted bucket-index* that stores mappings from the values of a dimension to relevant buckets. We illustrate such an index on the left in Figure 9. The references from values to buckets can be used during join processing to efficiently determine all regions that contain a certain

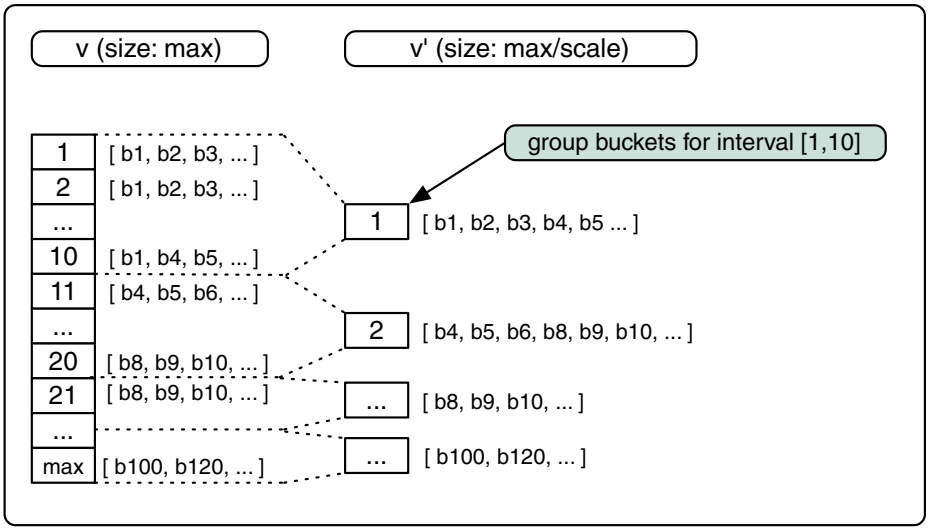



**Figure 9** Illustration of an inverted bucket index.

value. Note that this is the same principle as in a hash join. However, rather than applying a hash function to the join values, we only have to collect the references from dimension values (which are in fact, hash values) to buckets. For clarification and to differentiate from the general problem of hashing for the data summaries we call this join *index join*. A full index of that kind can result in a very high in-memory requirement. In the worst case—when all buckets span over the whole dimension range—for each dimension value we have to store the references to all buckets. To lower the memory requirements, we introduce an approximation by storing references for a range of values rather than single values. This is depicted in Figure 9 on the right, assuming that we use a scale factor of 10 (i.e., ranges of size 10: [1, 10], [11, 20]).

Ranges that are not covered by any bucket are omitted in the index. In an index without approximation we have to store $\sum_{R\in\mathcal{R}} \overleftrightarrow{R}^j = \overline{\mathcal{R}}^j \cdot |\mathcal{R}|$ entries, where $\mathcal{R}$ refers to the set of regions that have to be indexed, $\overleftrightarrow{R}^j := R[j].hi - R[j].low + 1$ to the range of region $R$ in join dimension $j$, and $\overline{\mathcal{R}}^j := \frac{1}{|\mathcal{R}|} \cdot \sum_{R\in\mathcal{R}} \overleftrightarrow{R}^j$ to the average range of all buckets from $\mathcal{R}$ in join dimension $j$. The size of the approximated variant cannot be determined exactly for the general case, as it depends on the actual distribution of regions in the whole range of $\mathcal{R}$. However, we can provide an upper limit as $O\left(\frac{\overline{\mathcal{R}}^j \cdot |\mathcal{R}|}{scale}\right)$, where *scale* refers to the used scale factor.

**Algorithm 4** Principle of index join.

**Input**: left input $\mathcal{L}$, right input $\mathcal{R}$
**Output**: join space containing overlapping buckets

```
idx = buildInvertedBucketIndex(R);
forall L ∈ L do
    O = idx.getOverlappingBuckets(L);
    forall O ∈ O do
        J.add(determineOverlap(O, L));
    end
end
return J
```

Algorithm 4 illustrates the principle of the index join. As known from hash joins, we choose the smaller of both input sets to build the inverted bucket-index. In the algorithm, without loss of generality, we assume that this is $\mathcal{R}$. The method `determineOverlap` has to be called for all pairs of overlapping regions, as we need the fraction of overlap to estimate the number of results for join ordering (Section 5.2.3) and source ranking (Section 5.3). The exact overlap has to be determined only once per overlapping pair of regions, no matter how much they actually overlap. This is assured by method `getOverlappingBuckets`, which returns all overlapping buckets at once using the before created index.

A lower limit of the number of operations using the index join is provided by $\Omega(|\mathcal{R}| + |\mathcal{L}|)$, while the worst-case upper limit is

$$O\left(|\mathcal{R}| + |\mathcal{L}| \cdot |\mathcal{R}|\right). \tag{3}$$

The exact number depends on the actual distribution of buckets in $\mathcal{R}$ and $\mathcal{L}$ and can be approximated by

$$\Theta\left(|\mathcal{R}| + |\mathcal{L}| - |\{L \in \mathcal{L} : L[j] \frown \mathcal{L} \sqcap_j \mathcal{R}\}| + \sum_{L \in \mathcal{L}: L[j] \frown \mathcal{L} \sqcap_j \mathcal{R}} |\{R \in \mathcal{R} : R \frown L\}|\right), \quad (4)$$

where $\frown$ refers to the overlap relation and $\mathcal{L} \sqcap_j \mathcal{R}$ to the range(s) where buckets from $\mathcal{L}$ and $\mathcal{R}$ overlap in join dimension $j$. First, we need to scan all $|\mathcal{R}|$ buckets to build the index, then we have to scan all $|\mathcal{L}|$ buckets from $\mathcal{L}$. Only for those buckets from $\mathcal{L}$ that are in the overlap with $\mathcal{R}$ (i.e., all $L \in L : L[j] \frown \mathcal{L} \sqcap_j \mathcal{R}$), we have to actually call `determineOverlap` for each overlapping $R \in \mathcal{R}$. Thus, the index join becomes more efficient with smaller sizes of $\overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j$ and smaller average bucket sizes $\overline{\mathcal{L}}^j$ and $\overline{\mathcal{R}}^j$. To make this more evident, we can provide an asymptotic upper limit assuming uniform distribution of the buckets in $\mathcal{L}$ and $\mathcal{R}$ and an un-approximated inverted bucket-index. Then, the *density* of a set of buckets $\mathcal{L}$ can be determined as

$$d_{\mathcal{L}} := \frac{|\mathcal{L}| \cdot \overline{\mathcal{L}}^j}{\overleftrightarrow{\mathcal{L}}^j}.$$

The density describes the fraction of points in the whole range $\overleftrightarrow{\mathcal{L}}^j$ of $\mathcal{L}$ that are covered by buckets from $\mathcal{L}$. Using the density of a set of buckets, we can approximate the upper limit for the sum from (4) as

$$\begin{aligned} O\left(\underbrace{\frac{d_{\mathcal{L}} \cdot \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j}{\min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}}_{|\{L \in \mathcal{L}: L \in \mathcal{L} \sqcap_j \mathcal{R}\}|} \cdot \underbrace{\frac{d_{\mathcal{R}} \cdot \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}{\min(\overline{\mathcal{R}}^j, \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j))}}_{|\{R \in \mathcal{R}: R \text{ overlaps with } L\}|}\right) \\ = O\left(\frac{\frac{|\mathcal{L}| \cdot \overline{\mathcal{L}}^j}{\overleftrightarrow{\mathcal{L}}^j} \cdot \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j}{\min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)} \cdot \frac{\frac{|\mathcal{R}| \cdot \overline{\mathcal{R}}^j}{\overleftrightarrow{\mathcal{R}}^j} \cdot \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}{\min(\overline{\mathcal{R}}^j, \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j))}\right) \\ = O\left(|\mathcal{L}| \cdot \underbrace{\frac{\max(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j)}{\overleftrightarrow{\mathcal{L}}^j}}_{\leq 1} \cdot |\mathcal{R}| \cdot \underbrace{\frac{\max(\overline{\mathcal{R}}^j, \min(\overline{\mathcal{L}}^j, \overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j))}{\overleftrightarrow{\mathcal{R}}^j}}_{\leq 1}\right). \end{aligned}$$

This shows that $|\mathcal{L}| \cdot |\mathcal{R}|$ is a very rough upper limit in (3). In fact, the amount of operations reduces significantly with decreasing $\overleftrightarrow{\mathcal{L} \sqcap_j \mathcal{R}}^j$, $\overline{\mathcal{L}}^j$ and $\overline{\mathcal{R}}^j$.

### *5.2.3 Join ordering*

Besides the actual join implementation, the second crucial aspect for achieving a good performance in join processing is the join order. In principle, other well-known optimisation techniques can be mapped directly to the problem of ordering

region joins. We just have to use an appropriate cost model. The crucial cost factor for region joins is the number of resulting buckets. Thus, for the time being, we implemented a greedy algorithm using a cost function that estimates the number of resulting join buckets, as shown in Algorithm 5. The input for the algorithm is a set of pairs of relevant buckets for triple patterns that can be joined, as determined by the triple pattern source selection. Based on the number of buckets that might result from a join between two triple patterns in the worst case, the algorithm chooses the cheapest join in a greedy manner. One could extend this simple cost model by statistics describing the distribution of buckets in order to estimate the actual number of comparisons for each join.

The optimisations discussed above are only of basic nature. The focus of this work lies on the general applicability of hash-based data summaries for querying Linked Data. As part of that, we also analyse the general benefit we gain from optimising region joins (Section 7), which is expected to form only a small part of the entire query processing overhead.

**Algorithm 5** Greedy algorithm for join ordering.

**Input**: set $J$ of pairs $\{\mathcal{L}, \mathcal{R}\}$ of triple pattern results to join
**Output**: list containing ordered joins

```
O =⊥; min = ∞;
while J ≠ ∅ do
    forall {L, R} ∈ J do
        if |L| · |R| < min then
            min = |L| · |R|;
            next = {L, R};
        end
    end
    O.add(next);
    min = ∞;
    J = J \ {next};
end
return O
```

## 5.3 Result cardinality estimation and source ranking

As source selection is approximate, the set of relevant sources will usually be overestimated, i.e., contain false positives. Please note that false negatives are impossible: any region where results exist are guaranteed to be covered by the buckets of the summaries. Moreover, some queries may actually be answered by a large set of sources, such that a focus on the most relevant ones becomes important. Both issues suggest to introduce a ranking for sources identified as being relevant for answering the query.

One approach to rank sources according to their relevance is to use the cardinalities provided by the data summary. The intuition is that sources that provide many results should be ranked higher than sources providing only a few results. Thus, the idea is to estimate the number of results $s_r$ that each source $s \in \mathcal{S}$ contributes to. The ranks are assigned to sources according to the values of $s_r$ in descending order.

If each QTree bucket $B$ provides an estimated cardinality $c_B$ and a list of associated sources $\mathcal{S}_B$, we could simply assume uniform distribution and assign $c_B/|\mathcal{S}_B|$ to each source of a bucket, while summing up over all buckets. In early tests we recognised that this ranks sources very inaccurately. A simple modification of the summaries, which results in constant space overhead, is to record the cardinality $c_B^s$ for each source contributing to 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 triple patterns 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:

$$s_r = \sum_B c_B^s$$

Line 6 in Algorithm 2 can be adapted by applying the formula separately for each source, while substituting $c_B$ by $c_B^s$, $c_L$ by $c_L^s$ and $c_R$ by $c_R^s$.

To provide an example, we assume that bucket $L_1$ from the first triple pattern in Figure 7 summarises 60 triples from a source $s_1$ and 40 triples from a source $s_2$. Further, bucket $R_1$ from the second triple pattern shall refer to 20 triples from source $s_2$ and 50 triples from source $s_3$. The ratio between overlap and bucket is $\frac{2}{7}$ for $L_1$, respectively $\frac{1}{4}$ for $R_1$, and the larger bucket $R_1$ has an extension of 40 in the object dimension. Thus, after the first join we rank the sources as follows:

1. $s_3$ : contributes to $s_1 \bowtie s_3$ and to $s_2 \bowtie s_3$ : $\frac{60 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} + \frac{40 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} = 8.93$ of the join results
2. $s_2$ : contributes to $s_1 \bowtie s_2$ and $s_2 \bowtie s_3$, and doubled to $s_2 \bowtie s_2$ : $\frac{60 \cdot \frac{2}{7} \cdot 20 \cdot \frac{1}{4}}{40} + \frac{40 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} + 2 \cdot \frac{40 \cdot \frac{2}{7} \cdot 20 \cdot \frac{1}{4}}{40} = 8.57$ of the join results
3. $s_1$ : contributes to $s_1 \bowtie s_2$ and to $s_1 \bowtie s_3$ : $\frac{60 \cdot \frac{2}{7} \cdot 20 \cdot \frac{1}{4}}{40} + \frac{60 \cdot \frac{2}{7} \cdot 50 \cdot \frac{1}{4}}{40} = 7.5$ of the join results

Note that we estimate the *contribution* of each source to the join result. Thus, for each pair of joined sources we count twice—one time for the left-hand side, one time for the right-hand side. The estimated cardinality for the join result is actually half of the sum over all sources, i.e., 12.5 in the example. The determined cardinalities for each source are stored in the resulting bucket $L_1 \bowtie R_1$. They are used in the same way for result cardinality estimation and source ranking after the second join, which is still a rough approximation but can already significantly improve query processing performance. In order to guarantee that we do not miss any relevant source, we cannot discard any of the sources, no matter how small the estimated contribution is. Remember that the uniform distribution is just an assumption to enable a cardinality estimation at all. The source ranking based on this helps to assess the importance of all relevant sources. The effect 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.

Due to the assumptions we make during ranking, sources providing a large number of triples will usually be ranked higher than smaller sources although both large and small sources can potentially contribute to the query result. However, as

we show in Section 7, source ranking based on cardinality works satisfyingly accurate for real-world sources.

There are several possible approaches to improve the ranking accuracy, e.g., by inspecting the importance for each join dimension separately and determining a combined rank in the end. A crucial question that has to be answered before concerns the target of the ranking. In our approach, we rank sources higher that likely contribute to many results of the join. Alternative approaches can be based on the popularity of the sources using, for instance, PageRank [52], HITS [41], and optionally external information from Web search engines. Another alternative is to directly rank the importance of the generated join results rather than the importance of the sources that contribute to them. Ranking in our approach is very important and represents an orthogonal research problem in itself. In Section 7 we show that the current approach already indicates the actual importance ranking of sources in a satisfyingly accurate manner.

## 6 Evaluation setup

Our experiments aim at providing insight into memory consumption, runtime efficiency and query completeness for various lightweight index structures for source selection. We describe the setup of our experiments, methods and the test data in the following.

### 6.1 Experiments

We benchmark core functions of the index structures:

- **Index build**: The index structures should be able to handle the insertion of RDF data streamed directly from a crawler. We measure the time and memory needed to insert a given number of statements. We expect that our results are very close to the theoretical complexity analyses.
- **Query time**: Ideally, the source selection of the index structure returns only relevant sources for conjunctive SPARQL queries in milliseconds. The experiment executes SPARQL queries with two implemented join operators—the nested loop and inverted bucket list operator—and with and without the join ordering optimisation. We measure the execution time for the different operators and the QTree and multidimensional histogram.
- **Source selection**: Hash-based data summaries, SLI and II return an estimation of sources relevant to answering a query. Due to the approximate nature of these indexes the returned sources are always a superset of the actually relevant sources. Our experiment is designed to measure the average number of estimated sources in comparison to the average number of actually relevant sources. We expect that the number of selected sources is higher than the number of actually relevant sources.
- **Query answering**: The query results should justify the expensive execution of a SPARQL query directly over live Web content (we highlighted the challenges that are involved in fetching the content for the selected sources; e.g politeness and also temporarily server problems). Considering a source ranking, a Linked Data query processor would ideally use only the top-k sources to assure a worst

case query time and still guarantee a certain degree of completeness of the results. We evaluates the quality of the source selection and the completeness of the query answers wrt. the top-k selected sources. The baseline results are derived from the test queries over the materialised data set.

### 6.2 Datasets and queries

We use real-world datasets collected from the Web via the LDSpider [44], a Linked Data crawler. Two data set were gathered with a breadth-first crawl starting from a well-connected RDF source with a content filter for sources that contain `application/rdf+xml` files. The large dataset *L* contains 3.1 m RDF statements from 15.7 k sources. Further, we refer in some of the experiments to a smaller data set *M* which contains the first 50% of the content of *L*.

We experimented with randomly generated queries corresponding to two general classes. The first class of queries consist of *"star-shaped queries"* with one variable at the subject position. The second type of queries are *"path-shaped queries"* with join variables at subject and object positions. Figure 10 shows abstract representations of these query classes. The query classes of choice are generally understood to be representative for real-world use cases and are also used to evaluate other RDF query systems (e.g., [50]).

The star-shaped queries were generated by randomly picking a subject URI from the input data and arbitrarily selecting distinct outgoing links. Then, we substituted the subject in each triple pattern with a variable. Path queries were generated using a random walk approach. We randomly chose a subject URI and performed a random walk of pre-defined depth. The result of such a random walk was transformed into a path-shaped join by replacing the connecting nodes with variables.

Using these approaches, we generated from the data 50 queries for each query class with no, one and two join operations. We use `P-`*n* to denote path queries with *n* join operations and `S-`*n* to denote star-shaped queries with *n* join operations.

### 6.3 Setup

We use a dual core AMD Opteron 250 server with 4GB of memory and two 1TB hard disks, running Ubuntu 9.10/x86_64 (latest stable release of Ubuntu) for

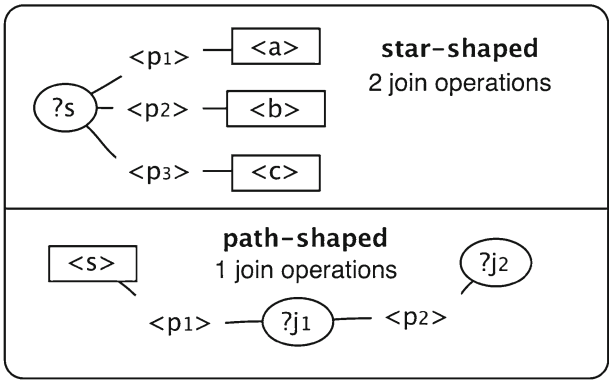



**Figure 10** Abstract illustration of used query classes.

our experiments. We base our experiments on implementations in Java, and use Version 1.6 of the Sun Virtual Machine.

We used reference implementations for the schema-level index and the inverted URI index:

- **Schema-level index (SLI)**: We use the standard Java HashMap implementation to store the list of source URIs that contain a property $p$ and/or the object (class) of an `rdf:type` statement. We arrive at two maps: one has as keys the RDF properties and as values a list of sources containing the corresponding property, and the second map has as keys the object of type statements and again as values the list of sources containing the corresponding type statements. To select relevant sources, we extract all RDF properties and classes contained in the query, perform lookups on the two maps, and return the union of the results of the lookups as relevant sources. Note, this does not involve any kind of ranking.
- **Inverted URI index (II)**: Our reference implementation of an inverted URI index also uses in-memory Java HashMaps, where the keys are the URIs in the dataset and the values are lists of source URIs which contain the key URI. We select relevant sources for a given query by extracting all URIs contained in the query, perform for each URI a lookup in the map and return the union of all lookup results. Note, this does not involve any kind of ranking.

The reason why we use reference implementation are the following: Our focus is not on the actual index and query performance, rather we are interested in how suitable these indexes are for the source selection and thus focus only on number of the resulting sources relevant to a query. The number of estimated sources is the crucial performance factor of our proposed query system as we will show in the evaluation section.

The datasets, queries and the implementations of the tested approaches are publicly available.[11] Thus, the presented evaluation is repeatable and extensible.

## 7 Results

In the following we present and discuss the results of our experiments. An overview of the concrete setup of our evaluation is given in Table 5. We provide for each experiments a selection of the results in plots, and compare and discuss the complete results.

### 7.1 Index build

The time needed for inserting a certain number of statements for MDH and QT is shown in Figure 11. The plot shows (with log-scaled xy-axis) the time elapsed to insert $n$-statements (*total* labelled point lines) for MDH and QT with different parameters.

We can see that MDH (two bottom lines) performs by order of magnitudes better than QT. The best insert time of MDH was achieved with string hashing with an average of 45 statements per millisecond. Among different QT configurations, mixed

[11] http://code.google.com/p/lidaq/

**Table 5** Overview of the setup of our experiments.

| | |
|---|---|
| Datasets | |
| $L$ | #stmt: 3.1 m, #src: 16 k |
| $M$ | #stmt: 1.53 m, #src: 6.6 k |
| Index types | |
| $QT_L$ | QTree($b_{\max}$: 50 k $m_{\max}$: 1 m $f_{\max}$:20) |
| $QT_S$ | QTree($b_{\max}$: 25 k $m_{\max}$: 1 m $f_{\max}$:20) |
| $MDH$ | Histogram($b_{\max}$: 50 k $m_{\max}$: 1 m) |
| $SLI$ | Schema-level index |
| $II$ | Inverted URI index |
| Hashing | |
| $MIX$ | Prefix and string hashing |
| $STR$ | String hashing |
| Queries | |
| S-$i$, $i = 0, 1, 2$ | 50 Star-shaped query with $i$ join(s) |
| P-$i$, $i = 0, 1, 2$ | 50 Path-shaped query with $i$ join(s) |
| Query planning | |
| $Op_N$ | Nested-loop join operator |
| $Op_I$ | Inverted bucket index join operator |

hashing performs best and needs in average 30 ms to insert a statement (or 0.04 statements per millisecond). Further, we observe that MDH increases the ratio of the inserted statements per time (cold start), whereas QT is getting slower. The differences between the insert times are due to the speed of the hash function in the

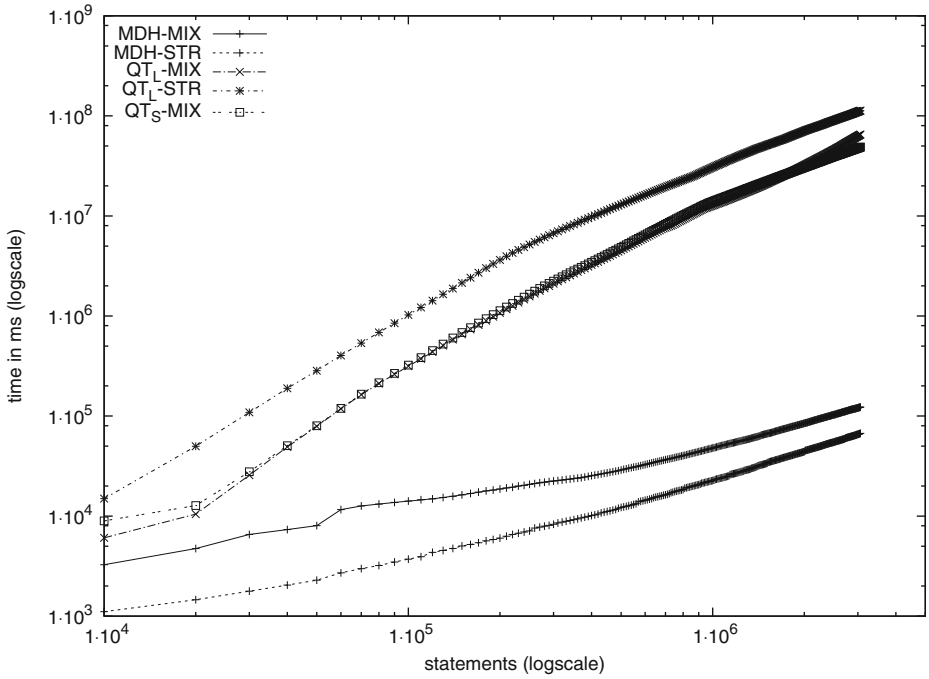



**Figure 11** Insert time per statements.

**Table 6** Index size and insert time of different approaches.

| Index version | Index size (MBytes) | Compression (%) | Index time (s) | Avg. stmts/ms |
|---|---|---|---|---|
| $QT_L$-$MIX$ $(L)$ | 42 | 7.4 | 65,147 | 0.04 |
| $QT_L$-$STR$ $(L)$ | 54 | 9.6 | 112,835 | 0.02 |
| $QT_S$-$MIX$ $(L)$ | 30 | 5.3 | 48,763 | 0.06 |
| $QT_L$-$MIX$ $(M)$ | 32 | 5.7 | 45,345 | 0.04 |
| $MDH$-$STR$ $(L)$ | 36 | 6.4 | 67 | 45.5 |
| $MDH$-$MIX$ $(L)$ | 11 | 1.9 | 122 | 24.40 |
| $SLI$ $(L)$ | 12 | 2.13 | 49 | 62.35 |
| $II$ $(L)$ | 49 | 8.7 | 56 | 54.30 |

case of MDH. The string hash functions are in general faster than the hash functions based on the prefix tree. In contrast, the string hashing functions significantly slow down QT. In general, string hashing equally distributes the string hashes over the allocated numerical space, which causes the QTree to recompute the bucket boundaries for nearly each inserted statements. Prefix based hashing clusters the input data more and reduces the operations to optimise the bucket boundaries in the QTree. (despite the fact that the computation time of the prefix hashing is slower than the string hashing). We can observe in all insert experiments that the at a certain stage we reach linear insert times (this corresponds with the theoretical analysis of insert complexity).

A complete summary of the results of the insert benchmark is presented in Table 6 (including the results for our reference implementation of II and SLI). The first column contains the index types and hash function as described in Table 5. The value in the brackets indicate which data set was used as an input. The column *compression* shows the fraction of the index size compared to the size of the raw data (561M for dataset L and 260M for dataset M). The table shows again the relatively poor indexing performance of QT. However, we will show that QT outperforms the other approaches in the quality of the source selection at the price of indexing time. Please not that the current implementation is a proof of concept, implemented with the standard Java data structures. We did not implement any low-level optimisations such as bit-based index structures or string encoding for the stored source URIs. For instance, we store full names of sources in each bucket together with the cardinality values. The general advantages with respect to the space complexity of the hash-based data summaries proposed in this work, i.e., scaling with the number of sources but not with the number of triples, are discussed in Section 3.2.1.

### 7.2 Query time

We first discuss the results of the join ordering optimisation and the various join operators in Figure 12 and second, we present total query times for different setups of the QTree in Figures 13 and 14.

As we highlighted already, we integrated a basic join ordering method which optimises the query processing. The results in Figure 12 show that join ordering has a huge impact for the runtime of the source selection. The values above the plot boxes show the time benefit of the ordering for the query execution. The values are relative to the query time without ordering, thus a negative value indicates that the query processing with join ordering is faster by the amount of the value; e.g path-shaped

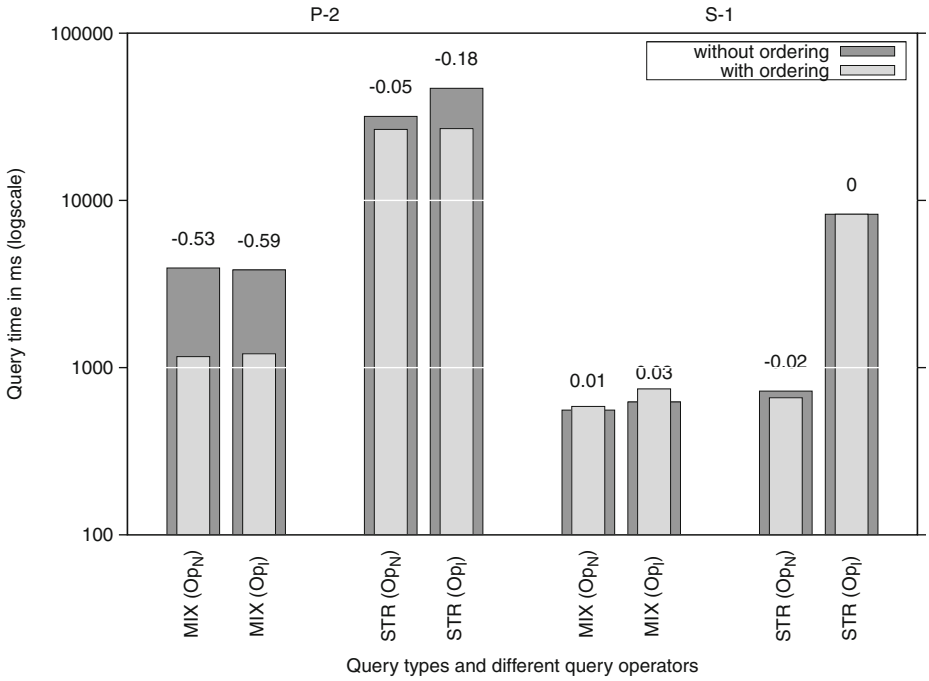



**Figure 12** Average query time with/out join ordering.

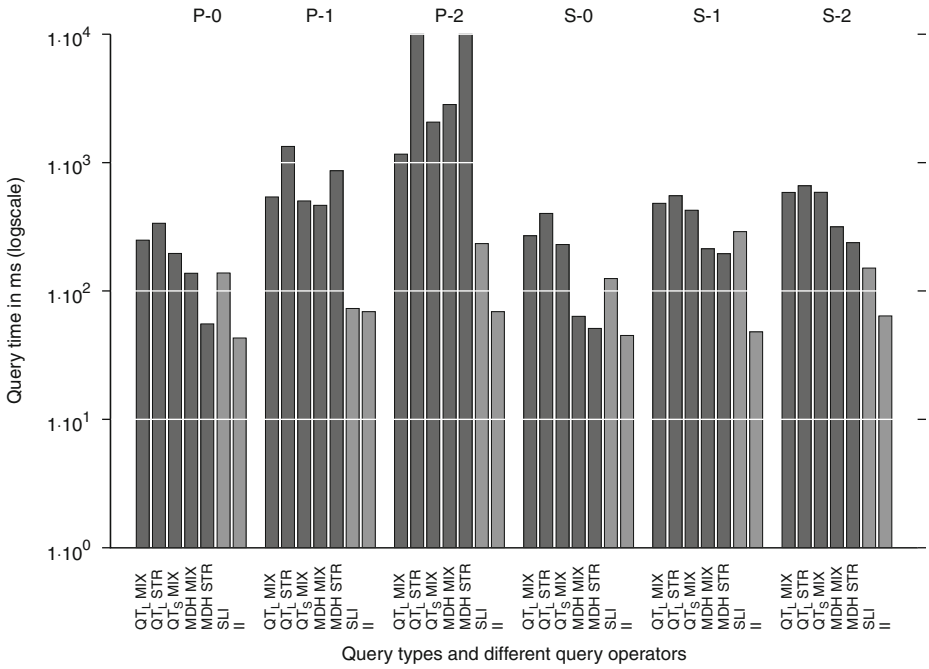



**Figure 13** Query time for different query types and approaches using a nested loop join operator.

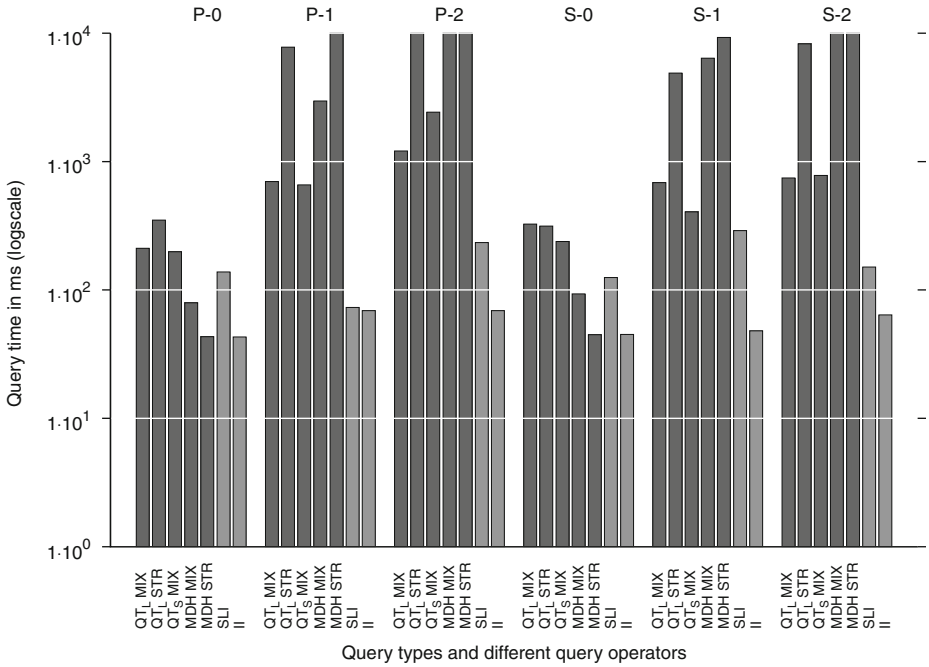



**Figure 14** Query time for different query types and approaches using a inverted bucket list join operator.

queries with two joins are 59% faster using join ordering and an inverted bucket list operator.

The plots show the average time needed to execute a query without and with the join ordering optimisation for the nested loop ($Op_N$) and inverted bucket list ($Op_I$) join operator. The selected index is $QT_L$ with hashing functions $MIX$ and $STR$. We omit in the plot the query runtime for other QTree or histogram versions for a better readability. However, we observed the following effects also for the other index versions. The big plot boxes show the query time without the ordering and the smaller light boxes the query time with ordering. The study of the results shows that join ordering achieves a large runtime benefit for path-shaped queries, whereas we cannot really see an optimisation effect for star-shaped queries. The difference is mainly due to the type of queries. Star-shaped queries contain constants at the predicate and object position which results in a more selective list of buckets, whereas our path-shaped queries have only one triple pattern with two constants and the other contain only one specified constant which makes join reordering more beneficial. We save more than 60% of the query time for path shaped queries with two joins for the QTree and over 50% for the star shaped queries with two joins.

Next, we present complete query runtime for different index versions, query types, join operators and our reference implementation of SLI and II. As expected, the reference implementations provide nearly constant query times outperforming the data summary indexes. The plots are limited on the y-axis to 10 s for better readability of the graphs. In addition, average query times of more than 10 s are not very

applicable in a real world scenario. The query times of the reference implementation are for all query classes less than 300 ms for SLI and less than 100 ms for II. We can see in Figure 13 that the average query time is less than 1 s for all query types using a nested loop join operator (beside the two outliers for path-shaped queries with two join variables and an index using a string hash function). Moreover, we can see that MDH is slightly faster than the QT variants. The expected benefit by applying a inverted bucket index join operator was not observed (Figure 14). We can see that this version of a join operators has query times for many of the query types of more than 10 s. Further, we observe an influence of the hashing function used; string hashing slows down query processing of QT and speeds up query processing of MDH.

In summary, we can say that the nested loop join operator outperforms the inverted bucket list join operator. II offer the fastest (and nearly constant) query times followed by SLI and MDH. QT shows the slowest performance in most of the queries.

However, these results about the actual source selection times are only one operation in the entire query processing and not the crucial factor. The most expensive part in evaluating a query is the dereferencing of the relevant sources via HTTP GET. Considering all the issues with accessing Linked Data (as highlighted in Section 2) we can expect that the number of estimated sources is the crucial factor.

### 7.3 Source selection

An important aspect is the quality and amount of the relevant sources estimated by the index structures. Figure 15 shows two interesting characteristics of the different index structures and query types and can be interpreted as follows: The difference between the number of estimated and real sources is the number of sources which are falsely selected as relevant. We show only the results for one setup of an QTree ($QT_L MIX$) with mixed hashing and a histogram with string hashing $MDHSTR$. These two setups return the best query answer completeness and thus, we decided to use them as the best representatives for the source selection evaluation. First, we see that the number of real query relevant sources are increasing with the number of joins for path-shaped queries and again, decreasing with the number of joins for the star-shaped queries. The same effect can be observed for the number of estimated sources for the path-shaped queries. As a surprise, the estimated number of sources relevant to answer the star-shaped queries is also increasing with the number of joins, which stands in contrast to the number of real query-relevant sources. Second, we find that QT shows the best performance in terms of the number of estimated sources to answer a given query. QT estimates the least number of sources for all types of queries. SLI, in contrast, estimates for nearly all query types the largest amount of sources.

Next we discuss the results of the query time to estimate the relevant sources and the number of the estimated sources itself. Let us consider an average lookup time of 500 ms to dereference a relevant source on moderate hardware that allows to perform 75 lookups in parallel (cf. [33]). Further, let us assume that all the returned sources are equally distributed over at least 75 domains which would allow us to perform the lookups in parallel. Thus, a round of 75 lookups would take in average 500 ms. From the results in Figure 15 we can see that QT outperforms the other

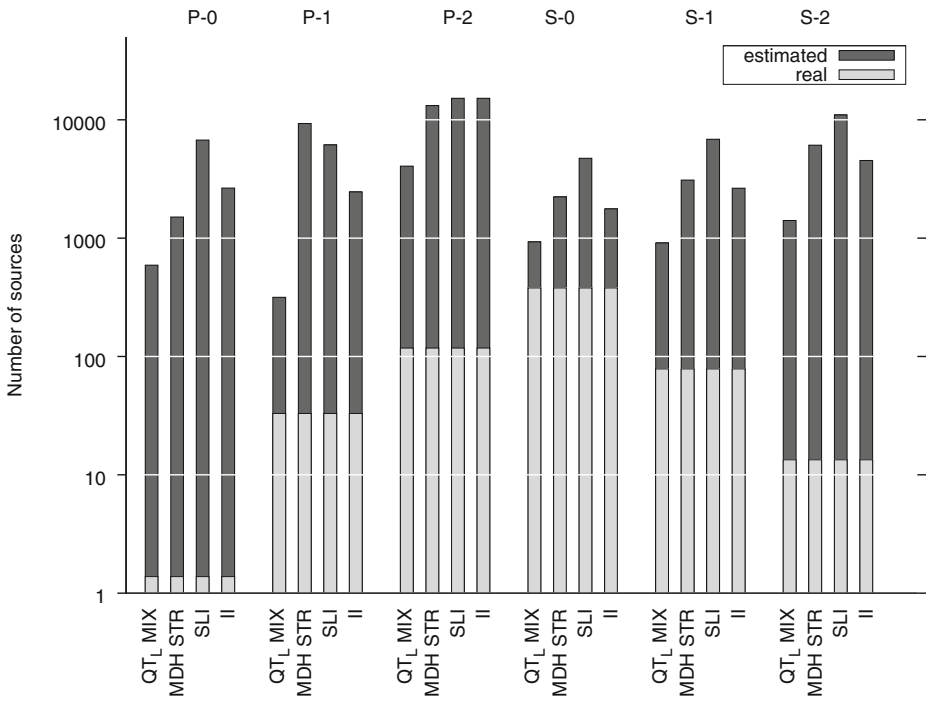



**Figure 15** Number of estimates and real query relevant sources (the *dark area* of the bars are the false positives).

approaches in the total number of estimated sources (for some queries by a factor of ten). The number of estimated sources for QT ranges between 600 and 1,200, which would require eight and 16 lookup rounds or a fetching time of 4–8 s. MDH, SLI, and II estimate for all query types at least 1,500 sources. Fetching the sources requires at least 20 lookup rounds or 10 s. The difference in the number of estimated sources slows down the query processing by at least 2 s. If we consider the query time to estimate the relevant sources and the time needed to fetch the content of the sources we can conclude that QT clearly outperforms the other approaches.

Eventually, the estimated time and performance of the source lookup operation is idealised and in a real-world setup one cannot assume that we can fetch all relevant sources in parallel while being polite at the same time. Thus, to be able to reduce the number of relevant sources we implemented basic ranking functions and evaluated these ranking functions next. Please note that the ranking also decreases the effects of false positives.

### 7.4 Source ranking

Querying all sources that were determined as relevant will still be too expensive in most cases for two reasons: (i) a huge number of sources may actually contribute to answering a query, where some of them contribute only minimally; and (ii) due to the approximation of the data summaries there may be false positives. For both

issues, ranking becomes crucial. Ideally, false positives and sources that contribute only few results are ranked lower than the sources that are actually relevant and can contribute the majority of results. Then, we can still provide a satisfying degree of completeness by querying only the top-k sources. The study of the top-k results presented in Figure 16 shows that QT significantly outperforms MDH. There are missing values for the histogram with mixed hashing for path-shaped queries with two joins (P-2) due to the fact that these queries timed out in our experiment (query timeout was set to 3 min). As our reference implementations for SLI and II do not support any kind of ranking and the returned sources are in random order, we omit the presentation of the results.

The source selection in QT returns over 40% of a query answer with the top-200 sources independent of the query type. MDH achieves only a query answer completeness of maximum 20% with the top-200 sources for all query types. We can observe that QT returns with the top-200 sources on average over 80% of the result statements for simple lookups with either the subject (S-0) or object (P-0) defined as a variable. For queries with one join (P-1,S-1) we still get over 60% of the results (with the top-200 sources). Moreover, our simple ranking algorithm yields 40% of the results with only the top-10 sources for simple lookups and queries with one join.

We can observe the same correlation between the hashing function and the index structure as in our other experiments. A string hashing favours the histogram and returns more results than a histogram with the mixed hashing. In contrast, the QTree performs better with the mixed hashing function than with a string hashing. Further,

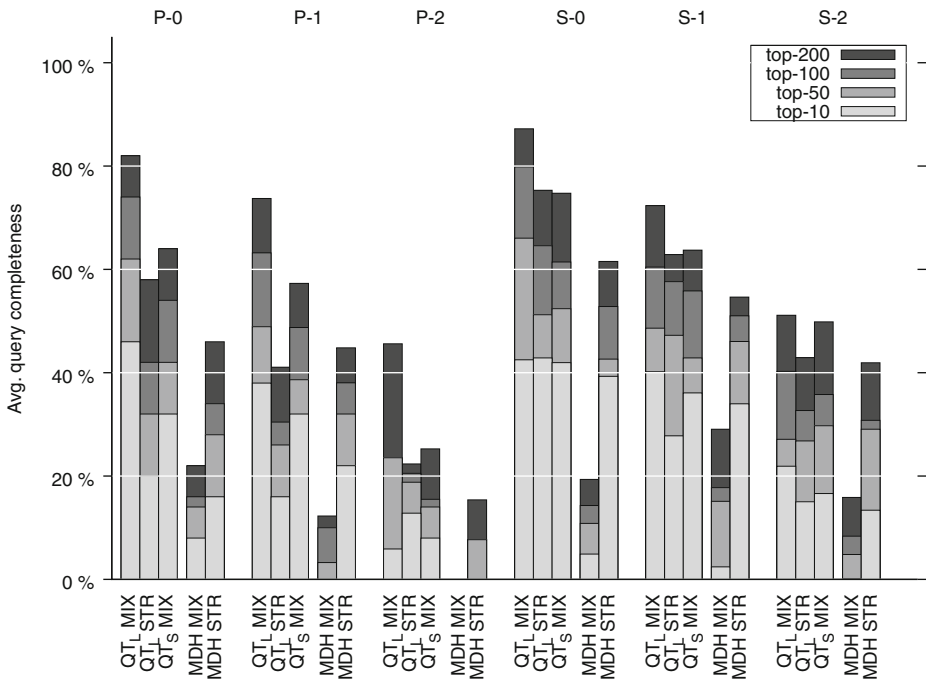



**Figure 16** Average query completeness.

we can see that the QTree version with more buckets performs better than with less buckets for the same input data.

Further, Figure 16 illustrates the influence of $b_{\max}$ on our approach. The general rule is that the more buckets we allow a QTree to use (higher $b_{\max}$), the less approximation has to be used in order to meet the $b_{\max}$ constraint. Therefore, the QTree provides more details in terms of smaller buckets and the number of false positives is reduced. However, by increasing $b_{\max}$, we also increase the required space in main memory. So, we need to find a tradeoff between false positives and index size. In general, the influence of $f_{\max}$ is negligible. As $b_{\max}$ and $f_{\max}$ are QTree specific parameters, discussing their influences in detail is out of the scope of this paper.

We decided not to perform the actual live lookup of the query processor since there are many factors involved which we cannot influence or ignore. One of the main factors are the time-dependent usage of the network and the available bandwidth which affect the download rate of the source contents. Another factor is the domain distribution of the selected sources which impacts politeness scheduling. In the best case we have $k$ distinct domains in the top-$k$ selected sources and can fetch the content with $k$ parallel lookups. In the worst case we have a single domain in the top-$k$ sources and need to perform $k$ sequential lookups with a wait time between each request.

### 7.5 Discussion

Based on our experimental results, we can state that the data summary approach, particularly the QTree, represents a promising alternative for querying Linked Data on the Web. The quality of the source selection of the QTree is superior to other approaches with regard to the number of returned sources.

The quality of the source selection and the fraction of possible answers is reasonably good using the QTree and rather poor for the other approaches. The QTree data structure outperforms the other approaches with the least number of estimated sources and the best query completeness. The basic query planning algorithms, together with the straightforward ranking algorithm, provide reasonably good query times and answer completeness for simple lookups and queries with one join operation. The QTree achieves a query answering completeness of over 40% (compared to global knowledge) with the top-200 sources throughout all our experiments (top value of 90% for star-shaped queries).

Regarding indexing times, the QTree data structure is the slowest approach compared to the alternatives. The difference in performance is due to the fact that MDH uses fixed and predefined bucket boundaries, whereas the QTree dynamically adjusts and optimises the boundaries based on the state of the inserted data. Our reference implementation of SLI and II require only to parse the necessary statements in the input file to build an index.

An ideal application scenario for the proposed data summary approach should have the following requirements: Firstly, it is not necessary to have complete answers, rather the application will return only the top-$k$ results. Secondly, the focus is on guaranteed up-to-date answers instead of possibly outdated results from old snapshots. Finally, the application allows a certain amount of time to execute and evaluate a given query.

The QTree is not designed to support bulk indexing of large amounts of data. However, the current insert times are reasonable in a very dynamic setup, in which an agent would index the content of Web sources either during runtime or only from a small number of sources at a time.

*Recommendations* In summary, we can clearly state that our approach—using data summaries like the QTree—provides the highest quality for the selection of query-relevant sources. However, as a price for the benefits, the aimed applications have to bear slower index times compared to other approaches, due to the higher complexity of the QTree as a summary. A crucial advantage of the proposed data summaries is that their size grows only with the number of inserted sources, and not with the number of inserted statements as it holds for the schema-level and inverted URI index.

Thus, for any application fulfilling the above mentioned requirements we generally recommend to use QT, due to its superior performance for source selection. QT is especially suited in a setup in which the indexing of sources is not performed in a bulk load process, but rather the sources are added one by one and in a setup with limited bandwidth and CPU capacity. QT and comparable data summary approaches are performing well on a large set of rather small sources, which makes such data summaries a suitable tool for enabling source selection on real-world Linked Data sources. In contrast, SLI and especially II show strengths in scenarios with few large sources, as for instance in a distributed index in a data centre setup. Further, in any scenarios where very low indexing and maintenance times are required, but querying a large fraction of actually irrelevant sources can be absorbed, SLI and II should be the tool of choice.

## 8 Related work

An implementation of the naïve Direct Lookup approach—iterative query processing with dereferencing bound URIs—has been recently presented by Hartig et al. [26]. As already sketched in Section 3, we believe our approach can be viewed as fruitfully expanding and generalising the 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 [38], primarily for selectivity and cardinality estimates over local data. The first histograms were only defined on one attribute value. In reality, we usually observe correlation between attributes and the assumption of independence often leads to bad approximations of result cardinalities [55]. The first histogram developed to counteract this problem is the two-dimensional equi-depth histogram proposed in [48]. First approaches for incremental maintenance of one-dimensional equi-depth and compressed histograms were proposed in [19]. However, most multidimensional histograms [22, 48, 55] are static and need to be reconstructed each time the data they summarise is updated. This is expensive and does therefore not serve our needs. Some [8, 61] even allow overlapping buckets, which are adapted during runtime. However, these approaches use one base bucket covering the whole data space to represent all the data that is not represented by separate buckets. The existence of this base bucket makes these approaches inapplicable to our approach

as it would always overlap coordinates we are looking for. Thus, we would always have to query all sources.

There exist several other summary techniques for large-scale multidimensional data that could be applied in this setup. There is a wide range of other hash-based structures, such as extendible hashing and linear hashing [37, 59]. The are other very interesting summary techniques that were mainly designed for the data-stream scenario, such as wavelets and sketches [3]. In data streams, these techniques are mainly used for approximate query answering, particularly for aggregation queries. Sketches represent a summary of a data stream using a very small amount of memory. They are typically used to answer distance queries, but also to estimate the number of unique values, e.g., for the estimation of the size of a self-join [3]. Gilbert et al. [20] shows how to use sketches to compute wavelet coefficients efficiently. However, there exist other methods to compute wavelets as summaries for data streams in a single pass. Chakrabarti et al. [10] shows how wavelets can be used for selectivity estimation and that they are usually more accurate than histograms. The authors show how to achieve approximate query processing by computing joins, aggregations and selections entirely on the wavelet coefficients. A main advantage of wavelets and sketches is that they are designed to support particularly efficient construction and maintenance phases. In contrast, a main advantage of the multidimensional structures discussed in this work is that they inherently capture data dependencies and were initially designed for multidimensional indexing, rather than approximate query answering on streams. Further, the purpose of this work is to highlight the benefits of data summaries in general, where the choice of the actual summary structure is a secondary matter. A detailed analysis of novel developments in the areas of multidimensional hash structures, sketches and wavelets will help us to identify which of these techniques are worth further investigation.

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 flat file containing small amounts of data. Stuckenschmidt et al. [63] 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 [43] 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)[12] 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 [50]. To answer queries with variables in any position of an RDF triple, RDF-3X holds indexes for querying all possible combinations of subject, predicate and object—an idea introduced in [24]. RDF-3X uses sophisticated join optimisation techniques based on statistics derived from the data. In contrast to our work, the approach (and similarly Hexastore [65]) uses a different data structure for the index and focuses

[12] http://rdfs.org/ns/void

on centralised RDF stores rather than distributed Linked Data sources. That said, a strategy which completely indexes RDF quads and performs lookups to validate results derived from the (possibly outdated) indexes is possible, however, involves overhead in creating, storing, and maintaining the complete indexes which we avoid with more lightweight index structures.

The inherent distribution of Linked Data natively suggests approaches for processing queries in a distributed manner as well. DARQ [58] is a federated query engine for SPARQL queries. Queries are decomposed into sub-queries and shipped to the corresponding RDF repositories. DARQ uses query rewriting and cost-based optimisations. It strongly relies on standards and Web service descriptions and can use any endpoint that conforms to the SPARQL protocol. Further, it does not require an integrated schema. DARQ supports standard query operators and two join variants on bound predicates. Query planning is static and centralised, utilising the service descriptions that replace any indexes. Sub-queries are distributed using a broadcasting mechanism.

Total distribution and decentralisation of query processing is a main target in the world of Peer-to-peer systems (P2P). In [29, 30] the whole RDF model graph is mapped to nodes of a distributed hash table (DHT). DHT lookups are used to locate triples. This implements a rather simplistic query processing based on query graphs that can be mapped to the model graph. Query processing basically conforms to matching query graph and model graph. RDF-Schema data are also indexed in a distributed manner and used by applying RDF-Schema entailment rules. On the downside, query processing has two subsequent phases and sophisticated query constructs that leverage the expressiveness of queries are not supported. There exist several other proposals for large-scale distributed RDF repositories [62, 63], RDF querying in Peer-to-peer environments [49], including approaches supporting partial RDF Schema inference [2]. RDFPeers [9] is another distributed infrastructure for managing large-scale sets of RDF data. Similar to, for instance, [40], each part of a triple is indexed, but whole triples are stored each time. Numerical data are hashed using a locality-preserving hash function. Load-balancing is discussed as well. Queries formulated in formal query languages, such as RDQL [47], can be mapped to the supported native queries. RDFPeers supports only exact-match queries, disjunctive queries for sets of values, range queries on numerical data, and conjunctive queries for a common triple subject. Query resolution is done locally and iteratively. Karnstedt [39] and Karnstedt et al. [40] extends this to sophisticated database-like query processing capabilities, total decentralisation, and considers also data heterogeneity. GridVine [1, 14] is a peer data management infrastructure addressing both scalability and semantic heterogeneity. Scalability is addressed by peers organised in a structured overlay network forming the physical layer, in which data, schemata, and schema mappings are stored. Semantic interoperability is achieved through a purely decentralised and self-organising process of pair-wise schema mappings and query reformulation. This forms a semantic mediation layer on top and independent of the physical layer. GridVine supports triple pattern queries with conjunction and disjunction, implemented by distributed joins across the network. It does not apply the idea of cost-based database-like query processing over multiple indexes. We see the distribution of query processing load as a promising way to achieve real scalability, data freshness and data authority. As such, investigating the usefulness of the QTree approach for decentralised setups is part of our agenda.

In fact, the QTree was developed in a P2P setup. Unstructured P2P systems leverage statistical data for source selection using so-called routing indexes. Crespo et al. [13] 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 [53], Bloom Filters [54], bit vectors [46], or the QTree. A common feature across these systems is to use a hash function to map string data to a numerical data space. In contrast to our work, the focus of query optimisation in P2P systems is to share load among multiple sites and on local optimisation based on routing indexes. However, a main problem in the current world of Linked Data sources is that they are usually quite restricted in query processing capabilities. All the decentralised approaches require SPARQL processing functionality at the participating nodes. As this is currently provided only by a handful of end points, we decided to focus on a centralised indexing approach that finally uses live queries as well, but on top of simple HTTP sources that do not provide advanced query processing capabilities. The mentioned decentralised approaches, however, bear high potential for designing scalable distributed index structures where single sources can connect to kind of super-nodes. These super-nodes form the actual index and are responsible for query routing and processing. An interesting work in this context with promising achievements focusing on Semantic Web technology is [60], which imposes a scalable and self-managing structured overlay on the participating nodes. This, together with the aforementioned proposals from the P2P area, will provide a perfect starting point for entering the domain of decentralised infrastructures for live queries on the Web of data.

## 9 Conclusion and future work

We presented an approach for determining relevant sources for live queries over RDF published as Linked Data. As these queries are issued ad-hoc, optimisation has to be done for all possible queries and cannot specialise on a specific type or a subset of queries. The presented approach uses hash functions to transform RDF statements into numerical space and data summaries to efficiently describe the data (RDF triples) provided by the sources. We discussed two variants of data summaries in conjunction with several hash functions and how to construct them. Furthermore, we discussed how to use these summaries to determine relevant sources for queries with and without joins. To limit query execution costs, i.e., the number of queried sources, we proposed the optional use of ranking to prioritise sources. In addition to theoretical analyses, we provided an extensive evaluation highlighting the influence of data summaries, hash functions, and query types on performance. These results show that our approach is able to handle more expressive queries and return more complete results to queries compared to previous approaches.

Based on this results we consider to combine our QTree approach with a sophisticated user interface that allows users to navigate and browse Web content. In general, user interfaces over Linked Data use conjunctive SPARQL queries for the user interactions. Further, these user interfaces do not require to show the complete set of answers for a given query in general; typically, they display only the top-k

results (similar to the front-ends of the traditional Web search engines). The current systems use materialised indexes as the underlying index structure which requires a significant amount of on-disk storage capacity and/or do not allow to use complex SPARQL queries. Our solution can be a very interesting alternative to the existing ones and will provide a lightweight application which still offers fast query times and reasonable query completeness over the top-ranked sources. Using live query evaluation over sources which check access control in a decentralised manner would allow for application of Linked Data in corporate environments.

Among other things, future work should consider how to optimise index construction. Especially in case of the QTree efficiency can be enhanced by no longer inserting RDF statements one by one but by applying pre-clustering techniques, i.e., by applying an efficient clustering algorithm to group statements in advance and then insert the clusters into the QTree. Pre-clustering is also relevant for other types of MDH that are better suited than equi-width histograms for non-uniformly distributed Web data. Such advanced MDH require a-priori knowledge about the actual distribution and suffer from increased costs for construction and maintenance. Another interesting aspect in this context is to develop an approach towards hashing that improves optimisation and evaluation of range queries. Future work will also have to consider the optimisation of queries involving joins. The bottleneck so far is that intermediate results can be huge. Thus, reducing the size of these intermediate results is one possibility to increase efficiency. Another one is to consider techniques for join optimisation that have originally been developed for distributed and parallel database systems, i.e., considering pipelining and parallelisation techniques for query processing. This goes hand in hand with other general questions of query planning and optimisation, such as choosing between several indexes available for the same graph pattern. We plan to investigate and specialise existing approaches from the database realm to further improve query planning and optimisation in our approach.

We would also like to investigate what aspects, in addition to the number of lookups, are relevant to improve the accuracy of source selection. Among others, this will involve a detailed analysis of the dynamics of Linked Data, e.g., using techniques from data mining and machine learning [64]. Moreover, integrating reasoning over the collected data would allow for returning consistent results adhering to the specified semantics.

In Section 8, we already highlighted the great potential that we see in distributed and decentralised query processing approaches. With the increasing availability of SPARQL endpoints on the Web, the idea of exploiting local query processing power from many machines becomes increasingly relevant and applicable to achieve Linked Data management and querying on true Web scale. Thus, we plan to investigate and explore these potential in the long term.

**Acknowledgements** This work has received support from the Science Foundation Ireland under Grant Nos. SFI/08/CE/I1380 (Lion-2) and 08/SRC/I1407 (Clique), and the European Commission under project ACTIVE (IST-2007-215040). We thank Aidan Hogan for comments.

## References

1. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Van Pelt, T.: GridVine: building internet-scale semantic overlay networks. In: ISWC'04, pp. 107–121 (2004)

2. Adjiman, Ph., Goasdoué, F., Rousset, M.-Ch.: SomeRDFS in the semantic web. JDS **8**, 158–181 (2007)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS '02, pp. 1–16 (2002)
4. Berners-Lee, T.: Linked Data, July 2006. http://www.w3.org/DesignIssues/LinkedData
5. Berners-Lee, T., Connolly, D.: Notation3 (N3): a readable RDF syntax, January 2008. W3C Team Submission. Available at http://www.w3.org/TeamSubmission/n3/
6. Bizer, Ch., Heath, T., Berners-Lee, T.: Linked data—the story so far. JSWIS **5**(3), 1–22 (2009)
7. Brickley, D., Miller, L.: FOAF vocabulary specification 0.91, November 2007. http://xmlns.com/foaf/spec/
8. Bruno, N., Chaudhuri, S., Gravano, L.: STHoles: a multidimensional workload-aware histogram. SIGMOD Rec. **30**(2), 211–222 (2001)
9. Cai, M., Frank, M.: RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In: WWW'04, pp. 650–657 (2004)
10. Chakrabarti, K., Garofalakis, M., Rastogi, R., Shim, K.: Approximate query processing using wavelets. VLDB J. **10**(2–3), 199–223 (2001)
11. Cheng, G., Qu, Y.: Searching linked objects with falcons: approach, implementation and evaluation. JSWIS **5**(3), 49–70 (2009)
12. Clark, K.G., Feigenbaum, L., Torres, E.: SPARQL Protocol for RDF, January 2008. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-protocol/
13. Crespo, A., Garcia-Molina, H.: Routing indices for peer-to-peer systems. In: ICDCS '02, pp. 23–32 (2002)
14. Cudré-Mauroux, P., Agarwal, S., Aberer, K.: GridVine: an infrastructure for peer information management. IEEE Internet Computing **11**(5), 864–875 (2007)
15. Cyganiak, R., Stenzhorn, H., Delbru, R., Decker, S., Tummarello, G.: Semantic sitemaps: efficient and flexible access to datasets on the semantic web. In: ESWC'08, pp. 690–704 (2008)
16. d'Aquin, M., Baldassarre, C., Gridinoc, L., Angeletou, S., Sabou, M., Motta, E.: Characterizing knowledge on the semantic web with Watson. In: EON'07, pp. 1–10 (2007)
17. Delbru, R., Toupikov, N., Catasta, M., Tummarello, G.: A node indexing scheme for web entity retrieval. In: ESWC 2010, pp. 240–256 (2010)
18. Garcia-Molina, H., Widom, J., Ullman, J.D.: Database System Implementation. Prentice-Hall, Englewood Cliffs (1999)
19. Gibbons, P., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. In: VLDB '97, pp. 466–475 (1997)
20. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In: VLDB '01, pp. 79–88 (2001)
21. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. In: VLDB'97, pp. 436–445 (1997)
22. Gunopulos, D., Kollios, G., Tsotras, V., Domeniconi, C.: Approximating multi-dimensional aggregate range queries over real attributes. In: SIGMOD '00, pp. 463–474 (2000)
23. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD '84, pp. 47–57 (1984)
24. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. In: 3rd Latin American Web Congress, pp. 71–80 (2005)
25. Harth, A., Hose, K., Karnstedt, M., Polleres, A., Sattler, K., Umbrich, J.: Data summaries for on-demand queries over Linked Data. In: WWW'10, pp. 411–420 (2010)
26. Hartig, O., Bizer, Ch., Freytag, J.-Ch.: Executing SPARQL queries over the Web of Linked Data. In: ISWC'09 (2009)
27. Hayes, P.: RDF semantics. W3C Recommendation, February 2004. http://www.w3.org/TR/rdf-mt/
28. Heimbigner, D., McLeod, D.: A federated architecture for information management. ACM Trans. Inf. Syst. **3**(3), 253–278 (1985)
29. Heine, F.: Scalable P2P based RDF querying. In: InfoScale'06, pp. 17–22 (2006)
30. Heine, F., Hovestadt, M., Kao, O.: Processing complex RDF queries over P2P networks. In: Workshop on Information Retrieval in Peer-to-Peer Networks (P2PIR'05), pp. 41–48 (2005)
31. Henzinger, M.R., Heydon, A., Mitzenmacher, M., Najork, M.: Measuring index quality using random walks on the web. Comput. Netw. **31**(11–16), 1291–1303 (1999)
32. Hogan, A., Harth, A., Umbrich, J., Decker, S.: Towards a scalable search and query engine for the web. In: WWW'07, pp. 1301–1302 (2007)

33. Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., Decker, S.: Searching and browsing linked data with SWSE: the semantic web search engine. Technical Report DERI-TR-2010-07-23, DERI (2010)
34. Hose, K.: Processing rank-aware queries in schema-based P2P systems. Ph.D. thesis, TU Ilmenau (2009)
35. Hose, K., Karnstedt, M., Koch, A., Sattler, K., Zinn, D.: Processing rank-aware queries in P2P systems. In: DBISP2P'05, pp. 238–249 (2005)
36. Hose, K., Klan, D., Sattler, K.: Distributed data summaries for approximate query processing in PDMS. In: IDEAS '06, pp. 37–44 (2006)
37. Huang, S.-H.S.: Multidimensional extendible hashing for partial-match queries. JPP **14**, 73–82 (1985)
38. Ioannidis, Y.: The history of histograms (abridged). In: VLDB '03, pp. 19–30 (2003)
39. Karnstedt, M.: Query processing in a DHT-based universal storage. Ph.D. thesis, AVM (2009)
40. Karnstedt, M., Sattler, K., Richtarsky, M., Müller, J., Hauswirth, M., Schmidt, R., John, R.: UniStore: querying a DHT-based universal storage. In: ICDE'07 Demonstrations Program, pp. 1503–1504 (2007)
41. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. JACM **46**(5), 604–632 (1999)
42. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4), 422–469 (2000)
43. Langegger, A., Wöß, W.: RDFStats—an extensible RDF statistics generator and library. In: Workshop on Web Semantics, DEXA (2009)
44. ldspider. Google code, April 2010
45. Manola, F., Miller, E.: RDF Primer. W3C Recommendation, February 2004. http://www.w3.org/TR/rdf-primer/
46. Marzolla, M., Mordacchini, M., Orlando, S.: Tree vector indexes: efficient range queries for dynamic content on peer-to-peer networks. In: PDP'06, pp. 457–464 (2006)
47. Miller, L., Seaborne, A., Reggiori, A.: Three implementations of SquishQL, a simple RDF query language. In: ISWC'02, pp. 423–435 (2002)
48. Muralikrishna, M., DeWitt, D.: Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In: SIGMOD 88, pp. 28–36 (1988)
49. Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmer, M., Risch, T.: Edutella: a P2P networking infrastructure based on RDF. In: WWW'02 (2002)
50. Neumann, Th., Weikum, G.: RDF-3X: a RISC-style engine for RDF. VLDB Endowment **1**(1), 647–659 (2008)
51. Oren, E., Delbru, R., Catasta, M., Cyganiak, R., Stenzhorn, H., Tummarello, G.: Sindice.com: a document-oriented lookup index for open linked data. IJMSO **3**(1), 37–52 (2008)
52. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation ranking: bringing order to the web. Technical report, Stanford Digital Library Technologies Project (1998)
53. Petrakis, Y., Koloniari, G., Pitoura, E.: On using histograms as routing indexes in peer-to-peer systems. In: DBISP2P, pp. 16–30 (2004)
54. Petrakis, Y., Pitoura, E.: On constructing small worlds in unstructured peer-to-peer systems. In: EDBT Workshops, pp. 415–424 (2004)
55. Poosala, V., Ioannidis, Y.: Selectivity estimation without the attribute value independence assumption. In: VLDB '97, pp. 486–495 (1997)
56. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF, January 2008. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/
57. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: ESWC'08, pp. 524–538, Tenerife, Spain. Springer (2008)
58. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: ESWC'08, pp. 524–538 (2008)
59. Rathi, A., Lu, H., Hedrick, G.E.: Performance comparison of extendible hashing and linear hashing techniques. SIGSMALL/PC Notes **17**(2), 19–26 (1991)
60. Schlosser, M., Sintek, M., Decker, S., Nejdl, W.: HyperCuP, hypercubes, ontologies, and efficient search on peer-to-peer networks. In: Agents and Peer-to-Peer Computing, vol. 2530, pp. 133–134. Springer (2003)
61. Srivastava, U., Haas, P.J., Markl, V., Kutsch, M., Tran, T.M.: ISOMER: consistent histogram construction using query feedback. In: ICDE '06, p. 39 (2006)
62. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., Houben, G.-J.: Towards distributed processing of RDF path queries. JWET **2**(2/3), 207–230 (2005)

63. Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., Broekstra, J.: Index structures and algorithms for querying distributed RDF repositories. In: WWW'04, pp. 631–639 (2004)
64. Umbrich, J., Karnstedt, M., Land, S.: Towards understanding the changing web: mining the dynamics of Linked-Data sources and entities. In: LWA 2010, FG-KDML, pp. 159–162 (2010)
65. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. VLDB Endowment **1**(1), 1008–1019 (2008)
66. Zinn, D.: Skyline queries in P2P systems. Master's thesis, TU Ilmenau (2004)