Towards Benefit-Based RDF Source Selection for SPARQL Queries

Katja Hose
Max Planck Institute for Informatics
Saarbrücken, Germany
hose@mpi-inf.mpg.de

Ralf Schenkel
Saarland University and MPI Informatics
Saarbrücken, Germany
schenkel@mmci.uni-saarland.de

ABSTRACT

The Linked Data cloud consists of a great variety of data provided by an increasing number of sources. Selecting relevant sources is therefore a core ingredient of efficient query processing. So far, this is either done with additional indexes or by iteratively performing lookups for relevant URIs. None of the existing methods takes additional aspects into account such as the degree of overlap between the sources, resulting in unnecessary requests. In this paper, we propose a sketch-based query routing strategy that takes source overlap into account. The proposed strategy uses sketches and can be tuned towards either retrieving as many results as possible for a given budget or minimizing the number of requests necessary to retrieve all or a certain fraction of the results. Our experiments show significant improvements over state-of-the-art but overlap-ignorant methods for source selection.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – Distributed databases; H.2.4 [**Database Management**]: Systems – Query processing

General Terms

Algorithms, Performance, Design

Keywords

SPARQL, RDF, Distributed Query Processing, Semantic Web

1. INTRODUCTION

The amount of Linked Data [3, 9] available on the Web has been continuously growing at a very high rate. The Linking Open Data (LOD) cloud[1] – a graphical representation of data published as Linked Data on the web – now covers 295 sources with over 31 billion data items from diverse domains. As more and more providers publish data, a growing amount of information will be represented multiple times in different sources. Already in the current LOD cloud, there are multiple providers for DBLP data, not counting the numerous other sources (ACM, IEEE, CiteSeer, etc.) that provide bibliographic information overlapping with DBLP. This effect will become even more important once semantically-enabled social applications allow everyone to publish their favorite movies, music, etc.

[1]http://lod-cloud.net/

An important consequence of the large number of available sources is a growing complexity of optimizing and executing SPARQL queries over linked data. As most sources would not have results for a query (or, more precisely, at least one of its triple patterns), carefully selecting sources is essential for efficient query processing. In existing systems, this is usually done either based on pre-computed information about sources [16, 18, 23, 28, 35] or `ASK` queries [29].

These approaches rely on the binary decision whether a source provides relevant data or not, none of them takes additional characteristics of sources into account, such as the fact that several sources may provide overlapping data, resulting in unnecessary requests that do not produce more results. There are further characteristics that are not taken into account, e.g., soft factors such as the quality of the data provided by a source, correctness, up-to-dateness, or fees for data access. Ideally, all this information should be considered for source selection. However, as a first step towards reaching this goal, this paper focuses on the aspect of overlap between sources and introduces *BBQ*, a strategy for **B**enefit-**B**ased **Q**uery routing for Linked Data.

We therefore propose a novel overlap-aware strategy for selecting sources for each triple pattern of a query. Our method uses an extended `ASK` operation that does not just return a boolean answer but retrieves a concise summary of the results in the form of Bloom filters. Based on these filters, it estimates the benefit of retrieving results for a triple pattern from a source, and ignores sources with low or zero benefit. Our source selection strategy can be tuned towards either retrieving as many results of a triple pattern as possible for a given budget or minimizing the number of requests necessary to retrieve all or a certain fraction of the results.

The remainder of the paper is structured as follows. We first give an overview of related work in Section 2 and then give important background information in Section 3. Section 4 introduces the extended `ASK` operation and Section 5 explains our benefit-based query routing technique. Finally, we present evaluation results for our method on a large data collection in Section 6 and conclude the paper in Section 7.

2. RELATED WORK

2.1 Source Selection

With the growth of the Semantic Web, efficient SPARQL query processing over multiple sources has attracted attention [15, 19] in the past couple of years.

However, the problem of selecting sources (i.e., SPARQL endpoints) that can contribute results to a SPARQL query has not been extensively covered. Some systems rely on manual annotations of queries with sources, which is possible with the `SERVICE` keyword in SPARQL 1.1 [2]. Other systems use pre-computed information and indexes collected during bootstrapping to decide on a source's relevance for a given query. This kind of additional information has been proposed in various flavors, e.g., service descriptions describing triple patterns that a source can answer [28], types of entities stored at a source [23], basic statistics such as the number of triples, subjects, predicates, etc. [1, 16], paths of predicates [32], frequent subgraphs [15,25,33] or one-dimensional histograms [22]. More complex summaries such as QTrees [18], multidimensional histograms [35], and RDFStats [22] also include information about the actual triples in the form of hashcode histograms of subjects, predicates, and objects. While algorithms for selecting a minimum number of sources have been proposed, e.g., selecting the top-k sources ordered by the expected number of results per source [35], none has yet considered the overlap between sources.

Using ASK queries [6, 29], on the other hand, in principle nothing needs to be pre-computed before the system can be used. Query processing runs in two phases. In the first phase, the sources are asked whether they can contribute to the query (binary answer), more precisely to the triple patterns contained in the query. Sources will only be considered for query execution (second phase) in case of a positive answer during the first phase.

The system outlined in [6], Avalanche, proposes to decompose a query into molecules (subqueries), identify relevant sources using a search engine, retrieve cardinalities for each unbound variable from the sources, combine molecules to possible query plans, and execute them. Avalanche is similar in spirit to the approach proposed in this paper but stops execution after the first k results haven been retrieved, whereas BBQ extends the ASK operation, considers the actual overlap between sources before query execution and is therefore able to estimate the number of results for combinations of sources prior to execution, execute parts of the query in parallel, and consider a recall-based stop criterion.

Source selection has been very popular outside the field of semantic data management. In distributed information retrieval, the problem is also known as query routing; among the many existing solutions are CORI [12], GlOSS [17], and ReDDE [31]. A number of recent methods takes inter-source overlap into account [7,20,30,31]. Shokouhi and Zobel [30] and COSCO [20] use the overlap of collections, computed independently of the query by sampling, to improve source selection for overlapping collections. Bender et al. [7] use Bloom filters as concise summaries of query results. All methods consider only keyword queries without further structure. Source selection has also been considered for deep Web sources [4] and keyword search on XML data [26].

Approximate query processing for RDF data has been considered by Tran et al. [34], but focusing on approximate matches of structural conditions (i.e., relationships of matching entities). In contrast, the approach proposed in this paper yields exact query results but may miss a few of them.

2.2 Summaries for Sets

There is a large number of data structures that provide concise approximate summaries of the elements in a set and provide methods for estimating the size of the intersection and/or union of two or more sets. Among the most simple approaches are histograms which model the distribution of values in a set but do not support the set operations well. Min-wise independent permutations [11, 24] estimate the similarity of two sets from the probability that both sets have the same minimal element under random permutation of the elements. Other distinct-value synopses store k minimal hash values (so-called *kmv synopses*) and estimate sizes of intersection and union of two sets from them [5]; they can also be extended by counters to support deletion of elements [8]. A number of methods map elements of the set to a bit vector and estimate the size of intersection and union of two sets from the number of bits in the binary AND and OR of the corresponding vectors, respectively. The most prominent among them are Bloom filters [10,27] that use k hash functions to improve estimation quality; we explain them in detail in Section 3.2. Literature proposes a number of variants such as counting bloom filters [14], spectral bloom filters [13], and bloom histograms [36].

3. BACKGROUND

This section discusses important background information on the approach proposed in this paper. We first introduce more formally the basics of RDF and SPARQL and then provide background information on Bloom filters, which our approach for source selection discussed in the following sections will make use of.

3.1 RDF and SPARQL

The Resource Description Framework[2] (RDF) was proposed by the W3C to represent information about subjects in a machine-readable format. Subjects are represented as URIs, and statements about them are made as triples of the form (`subjects`,`predicate`, `object`), where predicate is a URI and object is either a URI or a string literal. Definition 1 provides a more formal definition.

Definition 1. Given a set of URI references U, a set of blank nodes B, and a set of literals L, an *RDF triple* is a 3-tuple $t = (s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$ where s is called the subject of t, p the predicate, and o the object.

For example, the triple
(`<http://data.linkedmdb.org/resource/film/2982>`,
`<http://bbq.edu/producedIn>`,`2007`)
encodes that the movie identified by the URI
`http://data.linkedmdb.org/resource/film/2982`
(The Bourne Ultimatum) was produced in 2007.

We assume that the same entity is represented by the same URI in different sources; this assumption is implicitly made for many SPARQL federations [28, 29].

SPARQL[3] is a query language for RDF. At its core are *basic graph patterns* that consist of one or more *triple patterns*, which are in turn triples where each of the subject, property, and object may be a variable (denoted by a leading *?*). Shared variable names among different triple patterns express joins between triple patterns. SPARQL SELECT queries then retrieve bindings for (some of) the variables. Definition 2 formally defines the notion of a triple pattern and a basic graph pattern.

Definition 2. 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)*.

The value of variables can be constrained by additional FILTER expressions with operators such as $<$, $=$, and $>$ for numerical values and string functions such as regular expressions. Multiple graph patterns can be combined with a *UNION* operator.

The following example SPARQL query asks for Comedy movies produced in 2007:

[2]`http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`
[3]`http://www.w3.org/TR/rdf-sparql-query/`

```
SELECT ?s WHERE {
  ?s <http://bbq.edu/hasGenre> "Comedy" .
  ?s <http://bbq.edu/producedIn> "2007" .
}
```

It consists of two triple patterns, the first one defines that we are interested in Comedy movies, the second one defines that the movies in the result set have to be produced in 2007. The two triple patterns join on the subject position (`?s`). The output of the query is a set of URIs (bindings/mappings for `?s`) fulfilling these conditions.

In this paper, we are interested in the set of triple patterns $P(Q)$ appearing in a query Q. We denote the set of variables in triple pattern p as $V(p)$. We will further consider filter expressions in the query. As common for distributed approaches towards SPARQL processing, we restrict our considerations to acyclic basic graph patterns.

A SPARQL ASK query consists of the keyword `ASK` and a basic graph pattern, e.g.:

```
ASK {
  ?s <http://bbq.edu/hasGenre> "Comedy" .
  ?s <http://bbq.edu/producedIn> "2007" .
}
```

In contrast to a SPARQL SELECT query, the result is not a set of URIs but a boolean answer indicating whether the query has a solution (*true*) or not (*false*).

3.2 Bloom Filters

A Bloom filter [10] B for a set S of n elements consists of a bit vector of length m and k hash functions $h_i : S \mapsto \{1, \ldots, m\}$. For each element $e \in S$, all hash functions are evaluated and the corresponding bits in the bit vector are set. To test if an element e' is contained in S, all hash functions are evaluated for e'. If at least one of the corresponding bits is not set, e' cannot be contained in S; otherwise, it may still not be contained in S (i.e., it is a *false positive*), but this happens only with probability $(1 - e^{-kn/m})^k$ with $|S| = n$.

Given the bloom filters B_1 and B_2 for two sets S_1 and S_2 with the same hash functions and the same bit vector length, a bloom filter for the union $S_1 \cup S_2$ can be computed by a simple bit-wise disjunction of the bit vectors of B_1 and B_2; it is easy to show that this yields the same Bloom filter as if created directly from the union of the sets. For the intersection $S_1 \cap S_2$ of the two sets, an approximate Bloom filter can be created as the bit-wise conjunction of the two bit vectors; this is only an approximation since the same bit in both vectors may be set for different items and hence would not be set in a Bloom filter created directly for the intersection.

Even though Bloom filters were initially not meant for estimating the size of sets, it is possible to calculate a maximum likelihood estimate $\hat{S}^{-1}(t)$ for the number of elements in the corresponding set (the *cardinality* of the filter) when t bits in the bit vector are set (Papapetrou et al. [27]):

$$\hat{S}^{-1}(t) := \frac{\ln(1 - \frac{t}{m})}{k \cdot \ln(1 - \frac{1}{m})} \qquad (1)$$

We can immediately estimate the cardinality of the union of two Bloom filters with this formula, but not the cardinality of the intersection (since we can compute only an approximate Bloom filter here). Denoting the number of bits set in B_i by t_i and the number of bits in the intersection of two filters B_1 and B_2 by $t_\wedge$, Papapetrou et al. [27] show the following maximal likelihood estimate for the number $\hat{S}^{-1}(t_1, t_2, t_\wedge)$ of elements in the intersection of the corresponding sets:

$$\hat{S}^{-1}(t_1, t_2, t_\wedge) = \frac{\ln(m - \frac{t_\wedge \cdot m - t_1 \cdot t_2}{m - t_1 - t_2 + t_\wedge}) - \ln(m)}{k \cdot \ln(1 - 1/m)} \qquad (2)$$

There is no general formula to estimate the size of the intersection of more than two Bloom filters. In these cases, we first compute the conjunction of the bit vectors of each Bloom filter and estimate the cardinality based on that vector using Equation 1; this may overestimate the actual number of elements in the intersection.

4. EXTENDING THE ASK OPERATION

Existing systems for querying Linked Data often use a SPARQL ASK operation to identify possible sources that can contain results for a triple pattern p. For each triple pattern p in the original query, a query of the form `ASK {`p`}` is sent to each source which returns `TRUE` iff it can provide at least one result. The actual triple pattern query is then sent to all sources that replied with `TRUE`; multiple triple pattern queries can be batched [29]. To improve performance, results of these `ASK` queries can be cached and reused for future queries.

While this is a lean and efficient way of identifying sources with relevant results for a triple pattern, simple boolean answers are not very useful for query planning beyond excluding non-relevant sources. More specifically, it is impossible to identify sources that can only return results that were already retrieved from other sources. We therefore propose to extend the answer for an `ASK` query to include a sketch of the actual result of the triple pattern at the source. Such a sketch has two different components:

- a *count* of the number of results, possibly estimated from statistics available in the source's SPARQL endpoint
- a concise *summary* of the results, which is a lot more compact than actually returning all results (especially when there are many results)

Definition 3 formally defines a sketch for the result of a triple pattern.

Definition 3. The *sketch* returned by a source with triples T for a triple pattern p with variable set $V(p)$ consists of the pair $(c(p), S)$ where $c(p)$ is the number of triples from T matching p, and S contains for each variable $v \in V(p)$ a pair $(c(v), B(v))$, where $c(v)$ is the number of distinct bindings for v and $B(p)$ is a Bloom filter containing all bindings of v.

For triple patterns with two variables, Bloom filters for each variable's bindings are not always sufficient since they cannot represent which bindings were retrieved from the same triple. As an example, consider the pattern (`?s,actedIn,?o`). It is not sufficient to simply represent bindings for ?s and ?o independently since it would be impossible to reconstruct which combinations of bindings the source can provide. In this situation, our sketches additionally include a Bloom filter BC where combinations of bindings are represented. Note that all sketches of all sources need to use the same parameters for their Bloom filters; this restriction can be lifted with block-partitioned Bloom filters [27].

We do not specify a format for transmitting a sketch to the client but it would be easy to extend the existing SPARQL Query Results XML Format to include not just a boolean answer for `ASK` but also an encoded form of a sketch. Note that a source could provide sketches for frequently used triple patterns as some kind of meta data describing the collection, which eliminates the need for explicit `ASK` operations for these patterns. Similarly, sketches for frequent or all predicates could be pre-computed (triple patterns with a constant at the predicate position only), which can be used for predicting overlap of query patterns with this predicate but at lower precision than our query-based sketches for patterns with just one variable.

5. SOURCE SELECTION

In this section, we give a formal problem definition and discuss how these problems can be used for different types of SPARQL queries using BBQ (Benefit-Based Query routing for Linked Data).

5.1 Problem Definition

We consider a federation S of SPARQL endpoints or *sources* that partially overlap in their triples. Given a SPARQL query Q, we are interested in cheap execution plans based on the number of requests. In our cost model, a request corresponds to evaluating a triple pattern at a source. Further optimizations of the plan such as applying distributed join operators are orthogonal to this; this paper focuses on selecting sources for triple patterns, not on generating a good physical execution plan using these sources.

We focus on two different optimization problems that are relevant for different kinds of applications:

- Many applications require that all results for a query are found. If sources overlap in their triples, it is possible to achieve this by querying only a subset of all sources. Our first problem is therefore finding all results by querying a minimal number of sources.
- For other applications, the number of requests may be limited, or it may be sufficient to retrieve a subset of all results. Our second problem is therefore finding as many results as possible with a given number of requests (*maximal recall*), and our third problem is minimizing the number of requests to compute a certain number of results (*minimal plan*).

For a query consisting of a single triple pattern, the first problem is an instance of the set covering problem, which is known to be NP-complete [21]. In the same situation, the second problem is an instance of the maximum coverage problem, which is NP-complete, too [21]. As the first problem is a special case of the third problem, we will consider only problems two and three from now on. We will now formally introduce both problems.

We consider simplified query plans that determine only the sources at which each triple pattern is evaluated. We represent such a query plan as $qp \subseteq P(Q) \times S$; the *full plan* $FP(Q) = P(Q) \times S$ corresponds to evaluating every pattern at every source, even if the source does not include any matching triples for some patterns. We denote by $R(qp)$ the number of results of Q retrieved when evaluating the patterns in $P(Q)$ at the sources denoted in qp; $N(Q) := R(F(Q))$ denotes the number of results of query Q when evaluated with the full plan $F(Q)$, i.e., against all sources (precisely, when evaluating all patterns of the query at all sources and joining the results at a central node).

In general, evaluating Q with a plan $qp \subset F(Q)$ that does not evaluate all patterns at all sources may result in fewer results than with the full plan, i.e., $R(qp) \leq N(Q)$. We denote the recall of a plan qp as $RC(qp) := R(qp)/N(Q)$.

We can now reformulate our two problems; Definition 5 defines the minimal plan problem, Definition 4 the maximal recall problem.

Definition 4. Given a SPARQL query Q and a budget B for the number of requests, find a plan qp for Q such that $|qp| \leq B$ and $R(qp)$ is maximal under all plans with a size not exceeding B.

Definition 5. Given a SPARQL query Q and a target recall RC, find a plan qp for Q such that $RC(qp) \geq RC$ and qp has minimal size among all plans with this property.

We will now explain how the sketches introduced in Section 4 can be used for optimizing distributed execution of three kinds of queries: queries with single triple patterns, star-shaped queries of multiple triple patterns, and complex graph pattern queries.

5.2 Single Triple Patterns

We consider the case where p has a single variable; triple patterns with two variables can be treated similarly but use the Bloom filter BC for combinations of bindings. We solve the *Maximal Recall* problem using a benefit-based greedy heuristics for the maximum coverage problem; it can be shown that this achieves an approximation ratio of $e/(e-1)$ [21]. As a main difference, our algorithm does not operate on the true sets but on their Bloom filters.

Our algorithm first collects extended ASK sketches for p from all sources. We denote by $R(p, s)$ the Bloom filter for the result bindings when evaluating triple pattern p at source s. Our source selection algorithm runs in steps and chooses a new source for p in each step. R_i represents a Bloom filter for the result bindings after step i, and $R_0 = \emptyset$. The benefit $b_i(p, s)$ of choosing a yet unselected source s in step i is defined as the estimated number of unseen result bindings that s would contribute, or formally

$$b_i(p, s) := |R(p, s)| - |R(p, s) \cap R_{i-1}| \qquad (3)$$

Here, the size of the intersection is estimated from the Bloom filters as shown in Section 3.2. We choose the source s' with maximal benefit; the Bloom filter for the result set R_i after step i is computed as the union of R_{i-1} and $R(p, s')$. The algorithm stops after B steps. For the initial round, the subtrahend in Equation 3 is set to 0, i.e., in the initial round, we choose the source s' for which the Bloom filters indicate the maximum number of results.

We can apply the same algorithm also for the *Minimal Plan* problem, where we stop the algorithm as soon as the estimated number of results is at least as large as the target number of results. However, as Bloom filters may overestimate the size of the union of the per-source results, we may actually retrieve not enough results. In this case, we can continue the algorithm, using a Bloom filter of the actually retrieved results, and evaluate incrementally at the sources determined by the algorithm until enough results are found. This leads to increased processing times since we must access the additional sources sequentially but we expect that this happens only infrequently. For instance let us consider Table 1.

5.3 Star-Shaped Queries

A star-shaped query consists of multiple triple patterns with a single common variable on the same position (usually subject) of all patterns and constants or blank nodes otherwise. They retrieve entities that satisfy a number of conditions on their attributes. Examples include finding persons born in 1950 in Berlin (a simple star query with two triple patterns), or finding all female married actors born in Germany starring in the movie "Titanic" (a star query with four triple patterns, which could be the result of a faceted search interface). The latter example query would be represented by the following triple patterns (assuming the prefix `imdb` was declared before): `(?s,imdb:bornIn,dbpedia:Germany)`. `(?s,imdb:hasSpouse,[])`. `(?s,imdb:gender,"female")`. `(?s, imdb:actsIn,imdb:Titanic)`. Usually, there will be multiple sources that can return results for each pattern in a star-shaped query but a single source may not have enough information to answer the complete query, and results will usually be combined from the per-pattern result of different sources.

In this situation, we are not interested in finding all results for all triple patterns independently, but aim at finding results for the star-shaped query, i.e., entities that are in the result of each triple pattern. We therefore use a modified definition of benefit for star-shaped queries and a slightly modified version of the algorithm for source selection.

The algorithm again runs in steps, where in each step a pair of a source and a pattern to be evaluated at this source is determined. Thus, in each round the algorithm chooses the pattern-source pair (p, s) with the highest estimated benefit (i.e., the highest estimated number of unseen result bindings obtained by querying source s for pattern p). We denote by $R_i(p)$ the Bloom filter representing all results retrieved for pattern p after step i (computed as union of the filters from each source where p has been evaluated), and by R_i the result for the star-shaped query after step i (which is the intersection of the $R_i(p)$).

In step 0, we choose, for each pattern p, the source s_p where the estimated number of results is the highest. In step i, we determine the benefit $b_i(p, s)$ of evaluating pattern p at source s as the number of additional results that s contributes to the query result, or formally

$$b_i(p,s) := \left| \left((R_{i-1}(p) \cup R(p,s)) \cap \bigcap_{q \neq p} R_{i-1}(q) \right) - R_{i-1} \right| \tag{4}$$

After several rounds, we obtain a list of relevant sources for each pattern.

The benefit for querying a source s for a single triple pattern is determined by combining the Bloom Filters of already selected sources and estimating the result size with and without s. For star queries, this approach needs to be extended so that not only the Bloom Filters of one triple pattern are combined but also the Bloom Filters of all other already selected pattern-source pairs. The benefit itself is still the difference between the estimated number of results with and without the currently considered pattern-source pair. In the first few rounds, the algorithm will pick the most promising source for each involved triple pattern because only then bindings fulfilling all conditions can be obtained. Having selected a sufficient number of pattern-source pairs, the algorithm fetches the data for the triple patterns from the sources in parallel (time efficiency), evaluates the join, and outputs the results. For instance let us consider Table 2.

5.4 Complex Graph Patterns

Complex graph patterns contain at least two triple patterns and at least two different variables. A simple example is a query that asks for the production date of all movies where Tom Cruise appears, which could be formulated with the following two triple patterns: (`imdb:Tom_Hanks,imdb:actsIn,?m`). (`?m, imdb:producedInYear,?y`). It is impossible to accurately estimate the recall of such a query based on our sketches since they do not contain information about which years belong to which movies. The combined Bloom filters for the second pattern include this joint information, but they cannot be compared with the filter for the first pattern because the two filters contain different objects.

Fortunately, this is not a real problem as long as we want to compute all results for the query. In that case, we do not need an estimate for the results of the overall query. Instead, we can independently select a minimal set of sources for each triple pattern. Since this takes overlap into account, it will often select fewer sources than existing methods, while retrieving (almost) all results. This information is then fed into a query planner that can decide how to execute the query, for example which patterns to evaluate first, which joins to execute, etc. Here, the cardinality information retrieved in the sketches can be helpful. On top of this, our sketches can be used for two more optimizations: first, given two triple patterns p_1 and p_2 that join on a common variable, we can immediately decide which pairs of sources can deliver results that can be joined, and exclude any combinations that are guaranteed not to contribute to the join result. Consider, for example, two sources s_1 and s_2. If the intersection of the sketches is empty for the join variable for p_1 from s_1 and p_2 from s_2, their join will be empty. This can be especially helpful when it turns out that there are no join results across sources, since the query can then be evaluated as a whole at each source and their results combined (this is an extension of the *Exclusive Groups* strategy proposed in [29]). Second, we can use the sketches during query processing to eliminate potential sources. In the above example, we could first evaluate p_1 at all its selected sources, retrieving the set R_1 of bindings for $?m$. We now build the Bloom filter for this set. Before evaluating p_2, for example with an expensive distributed join, we first check all sources selected for p_2 if the intersection of their Bloom filter for $?m$ with the Bloom filter for p_1's results is empty, and discard a source if that is the case since it cannot contribute any results to the join. This incremental pruning of sources can largely reduce the number of sources accessed in a large query, even without source overlap.

ID	Triple Patterns	Explanation	Results
P1	1	movies produced in 2008	4248
P2	1	movies with actors playing themselves	7883
P3	1 with filter	movies with titles containing "Star Trek"	5
P4	1 with filter	movies produced in Germany	3878
P5	1 with filter	movies in black and white	23843
P6	1	movies with keyword "murder"	1771
P7	1	movies with keyword "goths"	1
P8	1	movies of Genre "Comedy"	177736
P9	1	movies of Genre "Romance"	4733
P10	1	movies with keyword "police"	823
P11	1	all female actors	40135
P12	1 filter	actors who died of heart attack	2408
P13	1 with filter	actors born in "Germany"	6124
P14	1 with filter	actors who died in the "USA"	15213
P15	1	actors born on "13 July 1940"	5
P16	1 with filter	actors born in 1980	1773
P17	1 with filter	actors who died in the 20th century	5929
P18	1 with filter	actors who died of cancer	3262
P19	1 with filter	actors born between 1900 and 1960	75805
P20	1	male actors	77053

Table 1: Single Triple Patterns

5.5 Filter Expressions

Our approach also supports filter expressions in queries. Whenever a filter condition affects only a single variable (such as `isURI(?o)`), it combines it with all triple patterns that contain that variable and adds the filter expression to the ASK statement sent to the sources. As a result, only triples matching both the pattern and the filter condition contribute to the sketch. Filters involving more than one variable are considered similarly in combination with a triple pattern if it contains all its variables. Some filter conditions are too complex to include in the source selection process, including filter expressions involving variables from different triple patterns (e.g., `sameTerm(?a,?b)`), since the per-source bloom filters do not provide enough details for precise estimations; we ignore such filters, which may lead to an overestimation of the number of results, but does not affect correctness.

6. EXPERIMENTAL EVALUATION

For our evaluation we used the IMDB dump from August 27, 2010. We extracted the first 100,000 movies and full information

ID	Triple Patterns	Explanation	Results
S1	2	movies produced in 2003 with an actor playing himself	390
S2	2	western movies with John Wayne	10
S3	2	romance movies with keyword "based-on-novel"	472
S4	3	western movies with keyword "based-on-novel" produced in the USA	80
S5	2	comedy movies produced in 2007	623
S6	2	documentary movies produced in 2008	848
S7	4 with filter	Sci-Fi movies produced in Canada, in German language, and in dolby digital	1
S8	2	romance movies with keyword "police" produced in India	29
S9	2 with filter	Black & white movies produced in the USA	13213
S10	2 with filter	Documentary movies about Arnold Schwarzenegger	4
S11	2 with filter	female actors born in Germany	1999
S12	3	real names of male actors who died of heart attack	172
S13	3 with filter	male actors born in France who died of cancer	68
S14	2 with filters	actors born in Germany who died in the USA	169
S15	2 with filters	actors born in Australia in the 1960s	210
S16	2 with filter	real names of actors born in 1980	569
S17	2 with filter	actors born in Greece and their movies	475
S18	4 with filters	actors born in the USA between 1900 and 1969 who died of heart attack in Germany	5
S19	2 with filters	actors born in 1900–1919 who died in 1960–1979	3045
S20	2 with filter	actors born in 1940–1949 and their movies	3533

Table 2: Star Queries

about all their actors and directors, about 46 million triples in total. Triples providing details about movies describe their year of production, country, language, cast, keywords, etc. Triples providing details about actors describe their birthdate, birthplace, date/place of death, cause of death, etc. In total, there are 46 different predicates for triples describing movies and 22 for actors.

6.1 Setup

To achieve controllable overlap of data sources, we grouped the triples and assigned the data to simulated RDF sources in a distributed setup. The two setups we focus on in this paper are generated by partitioning (i) movies according to their genre and (ii) actors according to their birthdate and birthplace. This means that all triples encoding information about a specific movie or actor are assigned to the same set of sources based on the partitioning condition. The movie setup resulted in 28 overlapping collections whereas the actor setup resulted in 22. Overlap in the movie setup results from the fact that many movies have more than one genre. Overlap in the actor setup exists because some sources provide information about actors selected by only one of the two criteria (birthdate and birthplace), others use both. Whereas the first setup has a "random" overlap where not all triples are stored as duplicates at different sources, the overlap in the second setup is more controlled and all information is guaranteed to be provided by 2 or 3 sources.

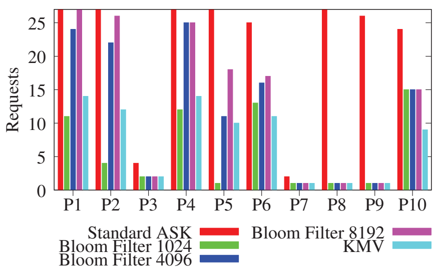
We consider single pattern queries and star queries. Since our approach considers triple patterns of complex queries independently (and query optimization with the advanced techniques outlined in Section 5.4 is beyond the scope of this paper), our evaluation focuses on these basic building blocks. Table 1 lists our single triple pattern queries (P1-P10 for the movie setup, P11-P20 for the actor setup), Table 2 lists our star-shaped queries (S1-10 for the movie setup, S11-S20 for the actor setup); both include the distinct result count for each query and a textual description of the query.

We mainly consider the number of requests of our method to collect all results, compared to a standard approach using `ASK` queries, i.e., sending `ASK` requests to all sources for all triple patterns and querying all sources with positive replies. Note that any method based on source statistics such as VoID would select the same sources since it cannot take overlap into account. We evaluate our method for different lengths of the Bloom filters' bit vectors, and additionally consider using KMV synopses [5] instead of Bloom filters to summarize results. We further analyze how recall grows when the number of allowed requests grows.

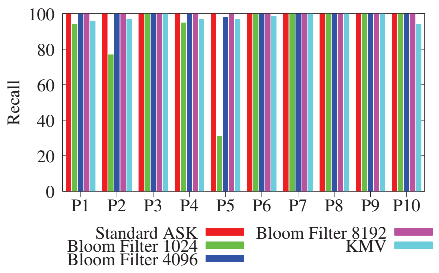
6.2 Results

We first consider single triple pattern queries. This case is especially important as it is the main building block for source selection with complex pattern queries. We configured the algorithm to retrieve all results with a minimal number of requests. Figure 1 shows the number of requests (1(a)) and the achieved recall (1(b)) for P1-P10 on the movie subcollections. It is evident that our algorithm requires a lot less requests than using standard `ASK` queries. At the same time, it achieves perfect recall for most queries with Bloom filters of at least 4096 bits. When the Bloom filter is too small, many results are mapped to the same bits and the resulting misestimations cause drops in recall. KMV synopses require slightly less requests but at the penalty of a slightly worse recall caused by misestimations.

The situation is very similar for queries P11-P20 on the actor subcollections, depicted in Figure 2. Here, the relative advantage for our method is even bigger, while preserving perfect recall. Turning to star-shaped queries, results are less clear. Figure 3 shows the number of requests (3(a)) and achieved recall (3(b)) for queries S1-S10 on the movie subcollections. Our methods again achieve a great advantage over `ASK` in terms of generated requests. However, even though recall is perfect for most queries when the Bloom filters are large enough, queries S1, S8, and S9 fall slightly behind. This can be explained by the fact that these queries contain either a pattern with many results (S1 and S8) or have a large number of results (S9), too many to be represented even with Bloom filters of 8192 bits. We expect that this problem will be mitigated using block-partitioned Bloom filters [27] that can adapt to the number of results. In addition, we expect that a user will be more likely tolerating a few missing results when the overall number of results is huge; in fact, she will probably refine her query anyway to retrieve fewer, more relevant results. Again, very similar results are achieved for queries S11-S20 on the actors subcollections, depicted in Figure 4. Finally, we turn to the analysis of the recall depending on the number of requests (the *Max-Recall* problem). For space reasons, we cannot discuss this in detail for every query, but have to focus on two typical example queries. Figure 5(a) shows number of requests and corresponding recall for the triple pattern query P10. It is evident that our methods select the good sources first, so the number of results grows quickly. Figure 5(b) shows the same for star-shaped query S3. Here, all methods retrieve almost 80% of the results with just the initial request that accesses one source for each

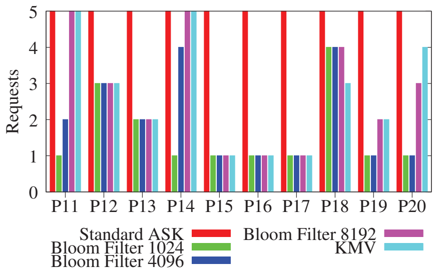


(a) Number of requests

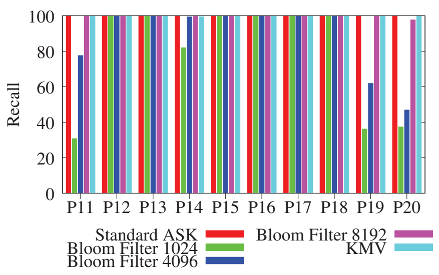


(b) Recall

Figure 1: Single Triple Pattern Queries P1-P10

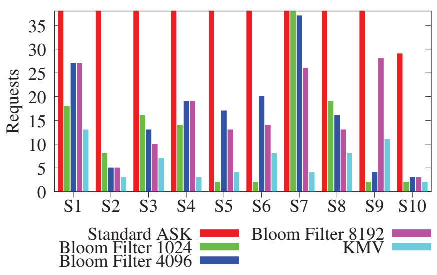


(a) Number of requests

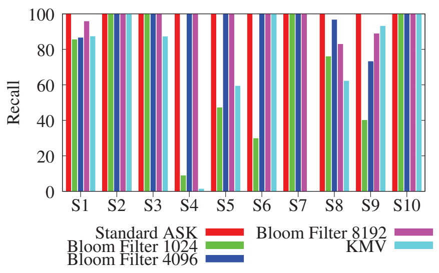


(b) Recall

Figure 2: Single Triple Pattern Queries P11-P20

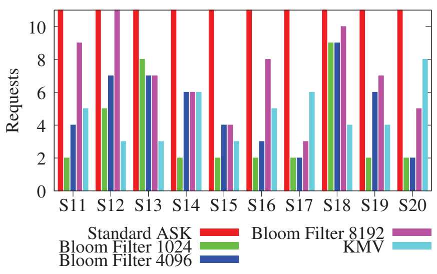


(a) Number of requests

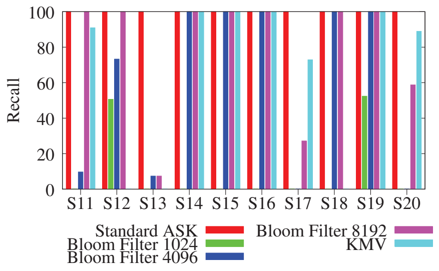


(b) Recall

Figure 3: Star-Shaped Queries S1-S10



(a) Number of requests



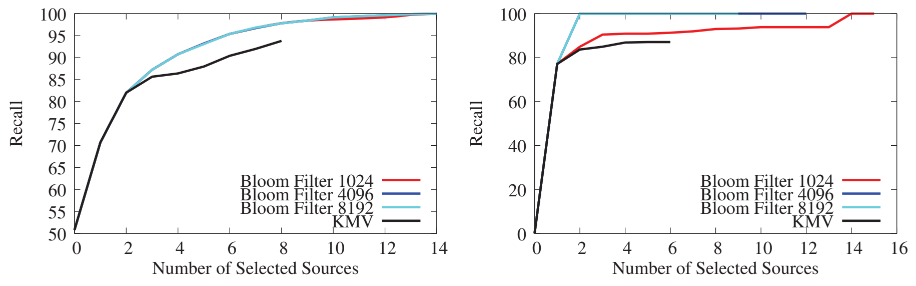
(b) Recall

Figure 4: Star-Shaped Queries S11-S20

triple pattern. With large enough Bloom filters, our method finds all results in the second step; when the filters are too small, estimation errors cause collections with fewer results to be accessed first. For both queries, KMV synopses underestimate the contribution of the remaining sources and therefore terminate early.

7. CONCLUSIONS AND FUTURE WORK

This paper presented an overlap-aware method for selecting a minimal amount of sources when querying linked data. A benefit-based greedy algorithm exploited sketches for each triple pattern of the query that were provided by the sources, using Bloom filters to



(a) P10 (b) S3

Figure 5: Request-Recall charts for queries P10 and S3

represent per-source results. The experimental evaluation showed clear improvements over overlap-oblivious methods while preserving perfect recall for almost all queries.

Our future work will exploit the Bloom filter-based sketches for query optimization and planning techniques for linked data. We will further consider Bloom filters of variable length and evaluate other possible set summaries.

8. REFERENCES

[1] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing Linked Datasets – On the Design and Usage of voiD, the Vocabulary of Interlinked Datasets. In *LDOW*, 2009.

[2] C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and optimization of the SPARQL 1.1 federation extension. In *ESWC (2)*, pages 1–15, 2011.

[3] S. Auer, J. Lehmann, and A.-D. N. Ngomo. Introduction to linked data and its lifecycle on the Web. In *Reasoning Web*, pages 1–75. 2011.

[4] R. Balakrishnan and S. Kambhampati. SourceRank: relevance and trust assessment for deep web sources based on inter-source agreement. In *WWW*, pages 227–236, 2011.

[5] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

[6] C. Basca and A. Bernstein. Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. In *SSWS*, 2010.

[7] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Improving collection selection with overlap awareness in P2P search engines. In *SIGIR*, pages 67–74, 2005.

[8] K. S. Beyer, R. Gemulla, P. J. Haas, B. Reinwald, and Y. Sismanis. Distinct-value synopses for multiset operations. *Commun. ACM*, 52(10):87–95, 2009.

[9] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[11] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[12] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *SIGIR*, pages 21–28, 1995.

[13] S. Cohen and Y. Matias. Spectral Bloom filters. In *SIGMOD*, pages 241–252, 2003.

[14] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[15] O. Görlitz and S. Staab. *Federated Data Management and Query Optimization for Linked Open Data*, chapter 5, pages 109–137. Springer, 2011.

[16] O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD*, 2011.

[17] L. Gravano, H. Garcia-Molina, and A. Tomasic. The effectiveness of gloss for the text database discovery problem. In *SIGMOD*, pages 126–137, 1994.

[18] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, pages 411–420, 2010.

[19] O. Hartig and A. Langegger. A Database Perspective on Consuming Linked Data on the Web. *Datenbank-Spektrum*, 10(2):57–66, 2010.

[20] T. Hernandez and S. Kambhampati. Improving text collection selection with coverage and overlap statistics. In *WWW (Special interest tracks and posters)*, pages 1128–1129, 2005.

[21] D. S. Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In *Approximation algorithms for NP-hard problems*, pages 94–143. 1997.

[22] A. Langegger and W. Wöß. RDFStats - an extensible RDF statistics generator and library. In *DEXA Workshops*, pages 79–83, 2009.

[23] A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *ESWC'08*, pages 493–507, 2008.

[24] P. Li and A. C. König. Theory and applications of *b*-bit minwise hashing. *Commun. ACM*, 54(8):101–109, 2011.

[25] A. Maduko, K. Anyanwu, A. P. Sheth, and P. Schliekelman. Graph summaries for subgraph frequency estimation. In *ESWC*, pages 508–523, 2008.

[26] K. Nguyen and J. Cao. K-Graphs: Selecting top-k data sources for XMl keyword queries. In *DEXA (1)*, pages 425–439, 2011.

[27] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality estimation and dynamic length adaptation for Bloom filters. *Distributed and Parallel Databases*, 28(2-3):119–156, 2010.

[28] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, pages 524–538, 2008.

[29] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *ISWC*, pages 601–616, 2011.

[30] M. Shokouhi and J. Zobel. Federated text retrieval from uncooperative overlapped collections. In *SIGIR*, pages 495–502, 2007.

[31] L. Si and J. P. Callan. Relevant document distribution estimation method for resource selection. In *SIGIR*, pages 298–305, 2003.

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

[33] T. Tran, P. Haase, and R. Studer. Semantic Search – Using Graph-Structured Semantic Models for Supporting the Search Process. In *ICCS '09*, pages 48–65, 2009.

[34] T. Tran, G. Ladwig, and A. Wagner. Approximate and incremental processing of complex queries against the web of data. In *DEXA (2)*, pages 171–187, 2011.

[35] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres. Comparing data summaries for processing live queries over linked data. *World Wide Web*, 14(5-6):495–544, 2011.

[36] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In *VLDB*, pages 240–251, 2004.