Supporting Identity Reasoning in SPARQL Using Bloom Filters

Gregory Todd Williams

Rensselaer Polytechnic Institute, Troy, NY, USA
willig4@rpi.edu

Abstract. The SPARQL Query Language presents a standardized interface to accessing RDF graphs. Its use allows templated queries to retrieve information from remote sources about known entities. However, such templated queries present a key problem: Queries cannot be made directly about nodes whose identity depends on reasoning. We present an efficient approach to solving this problem using a SPARQL extension function and the SPARQL XML Results format. We then discuss the use of this technique in federated query execution.

1 Introduction

The SPARQL Query Language[13] presents a standardized interface to accessing RDF graph data. It is particularly useful for querying remote endpoints to retrieve data about known entities. However, there are limits to how SPARQL may be used in this way: unlike entities identified by URI, queries cannot be made about specific blank nodes. Unfortunately blank nodes are often used in situations where they are uniquely identified by their properties. Such properties can be defined with the Web Ontology Language[9] (OWL) as being *Functional* or *Inverse Functional* properties. Blank nodes using these properties are endowed with identity, even if they themselves aren't directly identifiable.

Making queries about such blank nodes (with identities based on *Functional* and *Inverse Functional* properties) is desirable, even if SPARQL fails to support them directly. We present a system to allow efficient querying of identifiable blank nodes using Bloom filters[7], a probabilistic approach to set membership testing. A Bloom filter-based SPARQL extension function is used to determine which nodes satisfy the identity constraints of the query, returning a result set that closely approximates the desired data. Additional identity information about each result is returned with the result set, allowing the query results to be joined with local information to produce the desired information.

The rest of the paper is organized as follows. In section 2 we introduce a motivating example used throughout the paper, and discuss background information on formulating SPARQL queries for this example and the Bloom filter. In section 3 we introduce a Bloom filter SPARQL extension and show how it can be used to express equivalent queries. In section 4 we look at the space savings of using this approach in formulating queries (and by implication, the reduction

in query complexity). Section 5 discusses performance considerations relevant to deployment and optimization of this technique. Finally, in sections 6 and 7 we look at related work and conclude by proposing future work on optimization and analysis of this technique as well as studying its application in federated query environments.

2 Background

In this section we will examine the problem of querying a SPARQL endpoint for information about nodes identified by their properties. To do this, we use the following motivating example: we have a local address book database of known people we'd like to discover more information about. We would like to make a single query to a SPARQL endpoint, retrieving certain desired information (their phone number, for example) for all the people in our address book. Each person also has one or more identifying properties in the address book database (e.g. email address or homepage URL). Below we discuss how SPARQL can be queried for this information without the use of any extensions. We also look at Bloom filters, whose use will aid in more efficiently expressing and executing these queries.

2.1 Graph Pattern Matching in SPARQL

One possible solution to this problem is to load the remote data into the local store (merging it with the local identity data) and use an OWL reasoner before making the query. Unfortunately, this strategy can only work in situations where the remote data is accessible as RDF and the size of RDF to be loaded is reasonable for downloading, importing, reasoning and querying. In many common cases these requirements cannot be guaranteed; the data may be too large for downloading and local querying to be practical (such as DBpedia and DBLP from the Linked Open Data project[6]) or a SPARQL interface may be the only way to access otherwise private data. In these cases, using SPARQL is the only practical way to access the desired information.

In order to write a SPARQL query that will return the phone numbers of all the known people, the query must encode the identifying properties of each person. If we were only concerned with a single identifying property, such a query might look like this:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?email ?phone
WHERE {
  ?p foaf:mbox ?email ; foaf:phone ?phone .
  FILTER(
    ?email = <mailto:alice@example.org> ||
    ?email = <mailto:eve@example.net>
  ) .
}
```

Here we match all people with email addresses and telephone numbers, and use a filter to keep only those people who have an email address we know of. It is important to note that we must retrieve ?*email*, the email address used to identify the person, so that when we process the results we can match each telephone number with the appropriate person. The situation becomes more complex when we consider situations with more than one identifying property. For multiple properties, we must construct a pattern for each property and the associated values of interest, and take the union of all such patterns before going on to match for a telephone number:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?email ?homepage ?phone
WHERE {
  {
    ?p foaf:mbox ?email .
    FILTER(
      ?email = <mailto:alice@example.org> ||
      ?email = <mailto:eve@example.net>
    ) .
  } UNION {
    ?p foaf:homepage ?homepage .
    FILTER(
      ?homepage = <http://example.com/>
    )
  }
  ?p foaf:phone ?phone .
}
```

Since node identity isn't constrained to a single property-value pair, the number of potential patterns in the union can grow with the number of possible property chains (where the value of the first property-value pair is itself identified by another property-value pair).

This approach will work, but the query size scales with $O(nml)$, the number of local nodes (n) multiplied by the number of expected identifying properties per node (m) multiplied by the expected length of each property chain (l) with a constant multiplier of the expected size of representing the properties and values in each node's property chain in the SPARQL syntax (in this example, roughly 30 bytes are used for each equality test and single property-value pair).

As these examples indicate, the query size can grow quite large with just modest numbers of nodes and predicates. If possible, we'd like to express the query in a more compact form that decreases total bandwidth (both query *and* result size).

2.2 Bloom Filters

Bloom Filters[7] provide a probabilistic method for testing set membership. A Bloom filter consists of a bit vector of length N, and uses a set of t hashings

of an input to compute t bit addresses in the vector. The bits of the vector are initially set to all zero. For each member of the set, the bits of the vector referenced by the t hashings of the input are all set to one. Testing an input i for membership involves computing t bit addresses using the hashing functions on i, and checking each of the t referenced bits of the vector. If any of the t bits is set to zero, i is not part of the set. If all t bits are set to one, then i is accepted as belonging to the set. Notice that testing any valid member of the set will always involve t set bits yielding a positive result, while testing a non-member may involve t bits coincidentally set to one, resulting in a false positive.

The use of hashing techniques such as Bloom filters have been used in relational database systems as an efficient way to compute semijoins[8], [10]. This usage informs our approach of using them in SPARQL, a usage that may be seen as a semijoin over data whose equality may be tested based not just on node value (as in relational systems) but also on a node's relationships (the value of *Functional* and *Inverse Functional* properties).

3 Methodology

Here we present an extension function to SPARQL for testing a node's membership in a set based on a pre-computed Bloom filter. We have implemented this function in RDF::Query[14], a SPARQL implementation for Perl. The function, `<http://kasei.us/code/rdf-query/functions/bloom>` (abbreviated `k:bloom`), is invoked as `k:bloom( VAR, FILTER )`, taking a variable `VAR` and a serialized Bloom filter `FILTER` as arguments. The function returns true if the value bound to the variable is accepted as a member of the set, false otherwise. It may be used in place of the union-and-filter pattern described above with similar results (but with potential false positives).

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX k: <http://kasei.us/code/rdf-query/functions/>
SELECT ?p ?phone
WHERE {
  ?p foaf:phone ?phone .
  FILTER k:bloom(
    ?p,
    "AAAACgAAACIAAAACAAAAAgAAAAUAAAAD2aCcopHv9zsoAAQgADAuMg==\n"
  ) .
}
```

Fig. 1. SPARQL Query using a Bloom-filter function

In Figure 1 is an example of a SPARQL query using the Bloom filter extension function. Encoded in the Bloom filter are two nodes with the identifying properties:

– `foaf:mbox <mailto:willig4@rpi.edu>`
– `foaf:isPrimaryTopicOf`
  `<http://dblp.l3s.de/d2r/resource/authors/Gregory_Williams>`

This query will return bindings for ?p and ?*phone* for all nodes ?p that are identified with these specific `foaf:mbox` and `foaf:isPrimaryTopicOf` values (and might also, with fixed low probability, return false positives).

To encode all the information required for identity reasoning about each node, we must add each identifying property chain to the Bloom filter. To add each chain to the Bloom filter, we first encode each chain as a string based on the N3[5] syntax for paths. We refer to these identifying property chain strings as "names" for a given node. The name of a node is computed recursively as follows:

name(n) = "<" n ">" if n is an IRI
name(n) = "/" n "/" if n is a Literal
name(n) = "=" name(o) $\forall o$ s.t. { ?n `owl:sameAs` ?o }
name(n) = "!" p name(o) $\forall p, o$ s.t.
{ ?n ?p ?o . ?p a `owl:InverseFunctionalProperty` }
name(n) = "^" p name(o) $\forall p, o$ s.t.
{ ?n ?p ?o . ?p a `owl:FunctionalProperty` }

Thus, the two identifying properties above are represented with the names:

– "!foaf:mbox <mailto:willig4@rpi.edu>"
– "^foaf:isPrimaryTopicOf
  <http://dblp.l3s.de/d2r/resource/authors/Gregory_Williams>".

The error rate of a Bloom filter, or the probability that all t bits are set to one, is $P_e = (1-(1-\frac{1}{N})^{tc})^t$ with c being the number of items added to the filter. It can be seen that the error rate may be made arbitrarily small by increasing the size of the vector N. Using a fixed hashing count of $t = 7$ and error rate of $P_e = \frac{1}{100}$, for example, we see that the filter size grows by approximately 10 bits for each additional item expected in the filter. This space requirement is clearly better than the requirements of the union-and-filter approach mentioned in the previous section which depends on the length of the node identifiers and is on the order of at least tens of bytes per item.

Compared to the union-and-filter approach, the use of the `k:bloom` function doesn't leave us with a query that will return enough information to know which telephone number belongs to each person. This is because the query only returns values for ?p and ?*phone*, not any of the identifying property values such as email address. For this reason, the use of the `k:bloom` function triggers an extension to the query result generation code, allowing the required information to be returned along with the binding results.

During query execution, the implementation of the `k:bloom` function retrieves all the names for each candidate binding of `VAR` and tests them against

FILTER. If any name matches, the node is a member of the requested set with high probability. Each such matching node name is added to a list of "identity hints" that will be returned with the query results. Once results are returned to the client, the hints will allow the client to determine which local nodes share identity with each value of `VAR`.

The SPARQL XML Results Format allows a `link` element "with an `href` attribute containing a relative URI that provides a link to some additional meta-data about the query results"[4]. This `link` element is used to reference the list of "identity hints" generated during query execution. Two approaches to the use of this element are possible: either the identity information may be stored on the server with a persistent URL that the client may retrieve prior to joining the results with local data, or the identity information may be encoded using the `data:` URI scheme, allowing it to be included directly in the XML result data. Our implementation chooses to rely on `data:` URIs, but the next section discusses tradeoffs between the two approaches.

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
<head>
  <variable name="p"/>
  <variable name="phone"/>
  <link href="data:text/xml, (identity hints) " />
</head>
<results>
    <result>
      <binding name="p"><bnode>r1</bnode></binding>
      <binding name="phone"><uri>tel:+1-518-276-4431</uri></binding>
    </result>
</results>
</sparql>
```

Fig. 2. SPARQL XML Results with identity hints

Figure 2 shows an example of SPARQL XML results including a `data:` URL `link` element. For brevity, the actual identity hints content has been left out of the `data:` URL; the identity hints in this case might map:

(r1) ← "!foaf:mbox <mailto:willig4@rpi.edu>"

dictating that the blank node `r1` referenced in the results could be joined with any local node identified by the email address `<mailto:willig4@rpi.edu>`.

As is the case with Bloom filter use in relational databases, it is possible that due to the error rate of the filter, no local node will share identity with a value of `VAR`. In this case, there will be extra results and identity mappings. However, for each such extra result, the names of the node bound to `VAR` (provided by the

identity map) will not correspond to any local node names, and a join of the result and local data will simply discarded the extra results.

4 Results

Although we are currently working on a more thorough analysis of the use of Bloom filters in SPARQL queries (discussed in section 7), we present a brief set of results looking at the reduction in query size using Bloom filters in queries compared to equivalent union-and-filter queries.

We used the ESWC 2007 RDF data[1] as a basis for constructing queries. From this data, we consider three sets of resources which we use as the local data for Bloom filter construction: the set of all resources, the set of all `foaf:Person`s, and a small subset of people (chosen by matching an arbitrary substring against `foaf:firstName` values). We refer to these sets as large, medium, and small, respectively.

For each set, we constructed a query to retrieve a specific property value for each of the local resources using both the full union-and-filter-style query and a Bloom filter. Table 1 shows, for each set, the total number of local resources, the total number of identifiers (node "names"), and the corresponding query sizes (in bytes). We show the size for two different Bloom filter queries using different fixed error rates of 1% and 10%. In general, the choice of error rate (and thus filter size) will affected by the size of the SPARQL endpoint's database. If the endpoint's database is small, a larger error rate is acceptable as few false positives will be returned, while a large database will necessitate a smaller error rate to keep false positives small.

Table 1. Query size for Union-Filter (U-F) style and Bloom-filter (BF) queries (with 1% and 10% error rate)

Data Set	Resources	Identities	U-F Size	BF Size (1%)	BF Size (10%)
Small	23	47	3345	382	323
Medium	449	987	61274	1926	1095
Large	1419	1730	106176	3146	1705

As Table 1 shows, using a Bloom filter with a fixed error rate of 1% compared to an equivalent union-filter query, the query sizes shrink by between 88% and 97% for the small and large datasets, respectively. Whereas a union-and-filter query size of over 100 kilobytes for the large dataset is likely unreasonably large for a query over HTTP (the typical protocol for communicating with SPARQL endpoints), the equivalent Bloom filter query with 1% error (at 3146 bytes) is

[1] http://data.semanticweb.org/dumps/conferences/eswc-2007-complete.rdf

much more reasonable, even within the limit of most webservers for encoding the query in the request URL.

5 Performance Considerations

There are several performance issues to be considered when implementing and using a function such as the one described above. Here we discuss some of these issues and how they were dealt with in our implementation.

The use of Bloom filters as described here has the potential to reduce the overall bandwidth requirements of making a query. However, there are cases where the bandwidth requirements actually increase through the filter use. In cases where the selectivity of the filter on the remote data is close to 1, it may be more efficient to send a small query that selects all values, and simply remove unwanted data with a local join.

The use of the Bloom filter approach is predicated on the assumption that a typical SPARQL endpoint contains much more information than a typical query is interested in, and does not contain all relevant information about a node. In these cases, and with the ability to place an upper bound on the irrelevant data returned from a query due to Bloom filter error, we note that the expected total bandwidth required by the use of the `k:bloom` function is less than the previously described union-and-filter pattern resulting in an overall saving. Whereas the union-and-filter approach must send all names of local nodes in the (verbose) SPARQL syntax, by using `k:bloom` the same names can be sent in a single Bloom filter, and only those names that match remote data will be included in the result. Since the size of a filter given a fixed error rate is on the order of tens of bits per name (regardless of the size of the name), the space savings can be significant.

Beyond the space savings to be had through the use of Bloom filters, our intuition based on work in the relational database literature is that this approach may also yield performance gains based on query execution. We discuss this briefly below as potential future work.

Our implementation currently uses `data:` URIs for including the identity information in the query result. In choosing between using `data:` URIs and persistent URLs for storing the identity information, there is a tradeoff between code simplicity and query efficiency. Using `data:` URIs has the advantage of requiring no persistent storage for identity information, simplifying the endpoint implementation. The identity information is calculated during query execution (as each result is matched), and so isn't complete until the query execution is complete. Unfortunately, the `link` element of the result XML must appear in the document header before the binding data, requiring the entire result set to be held in memory until the `link` element is output. This might require a significant amount of memory at a performance cost to the server. Using a persistent URL for the identity information alleviates this memory requirement, allowing the result data to be streamed to the client but also requiring code to manage the data persistence and insure that persistent identity information isn't accessed

before all data is written (a possibility if the client requests the data concurrently as soon as the `link` element has been parsed).

Care must be taken in computing node names. Cycles in the property chains must not cause the computation to enter an infinite loop. Such situations may simply be ignored if no URI or literal value is included in the chain since such chains could never be used to identify a node in SPARQL. Even in cases where there are URI or literal values in a property chain, or simply in cases where a chain is particularly long, it may be desirable to declare a threshold length beyond which computation of names simply stops. This may result in queries that return in only partial results, a situation whose acceptability must be weighed against the computational demands and the specifics of the particular application. In RDF::Query, we do not define a cutoff threshold, favoring query completeness over efficiency.

To alleviate the issue of determining a cutoff threshold, notice that the names of nodes can be pre-computed, leaving only the actual hashing of the names (which depends on the query-specific Bloom filter) for execution at runtime. In cases where an endpoint's data is fairly static with infrequent updates, pre-computing the names of nodes could dramatically reduce the per-query cost of the `k:bloom` function. The cost of pre-computing the names would be amortized over all executed queries. Conversely, if the endpoint updates its data frequently, the number of total node names exceeded the practical ability to compute or store them, or the expected number of node names required by queries is much smaller than the total number of node names, it may be more desirable to compute node names at query execution time. Again, in RDF::Query we favor simplicity over efficiency and so compute names during query execution.

6 Related Work

Hashing has been used in relational databases to efficiently compute joins[8]. Mackert and Lohman[10] discuss the use of Bloom filters in distributed queries and find that the Bloomjoin outperforms many other distributed join methods except in cases where the involved semijoin has a selectivity very close to 1. Mullin[11] extends the Bloomfilter approach by sending a sequence of Bloom filters until sending back all remaining tuples is more efficient than continuing to filter them.

The ARQ SPARQL engine[3] includes a SPARQL extension to allow basic federated queries[1]. The extension can be used to make similar queries to those discussed in this paper, but is implemented using a naïve nested loop join and addresses neither the issue of identity reasoning nor of reducing the number of queries sent to the remote endpoint.

SPARQLfed[12] is an extension to SPARQL allowing intermediate result sets to be included alongside a query with a `BINDINGS` keyword, constraining the possible values of specific variables. The motivation of expressing variable binding constraints is shared between SPARQLfed and our work. Use of `BINDINGS` could be used to replace filtering in the union-and-filter pattern, but would still yield

very large queries and requires changes to the SPARQL grammar (a more complex change to a SPARQL engine than our extension function).

DARQ[2], an extension to ARQ, provides a more complete system for federated queries, including methods to define service descriptions including remote database size and the expected selectivity of triple patterns. Such values could be used to determine when Bloom filter use is appropriate, and to optimize the construction of Bloom filters. However, DARQ, like ARQ, fails to address identity reasoning in the evaluation of federated queries.

7 Future Work and Conclusion

The Bloom filter function introduced here could be used in federated queries similar to their use in relational database systems. RDF::Query implements the `SERVICE` extension (introduced in ARQ), and we modified the query engine to automatically use the Bloom filter function when making remote query calls. The modified implementation works as expected, interacting with both standard and `k:bloom`-enabled endpoints. Based on work with relational databases and Bloom filters such as [10], we believe their use may yield performance gains in federated query evaluation compared to the union-and-filter approach.

We are currently evaluating the performance of our system, and investigating how different optimization techniques affect query execution time. In particular, we hope to look more closely at the impact of traditional indexes on the union-and-filter approach compared to the impact of pre-computation on the Bloom filter approach, and to develop techniques for determining which will perform better for a specific query and dataset.

The federated query implementation in RDF::Query also has several restrictions we would like to remove. Currently, the use of Bloom filters in `SERVICE` calls rely on the join computing node names using local data. This requires all information used to compute the names of nodes used in a join (and so appearing as an argument to `k:bloom`) to be loaded in the local graph. This restriction doesn't preclude the join of two `SERVICE` calls, but the identity of nodes shared between such calls is always computed locally and so can't rely on remote knowledge of identity. Further work is needed to investigate how a query engine might choreograph the movement of result *and* identity data between remote endpoints to allow all participating endpoints to contribute to node identity computation.

The use of `k:bloom` works on the assumption that both the local client and the remote endpoint share an understanding of what properties are defined as *Functional* and *Inverse Functional*. We currently assume both sides have relevant ontologies loaded so that such properties can be used in computing node names. Future work in this area might allow the query to specify which ontologies, predicates, or even specific property chains should be considered in computing names. This information could be passed as additional arguments to the `k:bloom` function, but this runs the risk that the remote endpoint might need to load additional ontologies at runtime. Further study is needed to determine if

such a feature is needed in practice, and what level of granularity is appropriate for restricting the predicates used.

We have proposed the use of Bloom filters in SPARQL, allowing compact queries to be made about known entities. We have shown how this method can support joining query results with local data not just with direct node equality, but also with simple identity reasoning. The methods proposed in this paper have been implemented in RDF::Query and are available with that package from http://search.cpan.org/dist/RDF-Query/. We found the method to be useful in both simple and federated query evaluation, and are currently working on a more thorough analysis of theoretical and practical impacts of the method on query evaluation.

Acknowledgements

We wish to thank Sibel Adalı for her helpful comments on this work.

References

1. ARQ - Basic Federated SPARQL Query. http://jena.sourceforge.net/ARQ/service.html.
2. DARQ - Federated Queries with SPARQL. http://darq.sourceforge.net/, 2006.
3. ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/, 2008.
4. Dave Beckett. SPARQL Variable Binding Results XML Format. http://www.w3.org/TR/rdf-sparql-XMLres/.
5. Tim Berners-Lee. Notation 3. http://www.w3.org/DesignIssues/Notation3, 1998.
6. Chris Bizer, Tom Heath, Danny Ayers, and Yves Raimond. Interlinking Open Data on the Web. Demonstrations Track, 4th European Semantic Web Conference (ESWC2007), Innsbruck, Austria., 2007.
7. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
8. Kjell Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.
9. Mike Dean and Guus Schreiber. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/.
10. Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 149–159, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
11. James K. Mullin. Optimal Semijoins fo Distributed Database Systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.
12. Eric Prud'hommeaux. Federated SPARQL. http://www.w3.org/2007/05/SPARQLfed/, 2007.
13. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
14. Gregory Todd Williams. RDF::Query. http://search.cpan.org/dist/RDF-Query/.