Massive-scale RDF Processing Using Compressed Bitmap Indexes

Kamesh Madduri and Kesheng Wu

Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA 94720

Massive-scale RDF Processing Using Compressed Bitmap Indexes

Kamesh Madduri and Kesheng Wu
Lawrence Berkeley National Laboratory, Berkeley CA 94720, USA
{KMadduri, KWu}@lbl.gov

May 26, 2011

Abstract

The Resource Description Framework (RDF) is a popular data model for representing linked data sets arising from the web, as well as large scientific data repositories such as UniProt. RDF data intrinsically represents a labeled and directed multi-graph. SPARQL is a query language for RDF that expresses subgraph pattern-finding queries on this implicit multigraph in a SQL-like syntax. SPARQL queries generate complex intermediate join queries; to compute these joins efficiently, we propose a new strategy based on bitmap indexes. We store the RDF data in column-oriented structures as compressed bitmaps along with two dictionaries. This paper makes three new contributions. (i) We present an efficient parallel strategy for parsing the raw RDF data, building dictionaries of unique entities, and creating compressed bitmap indexes of the data. (ii) We utilize the constructed bitmap indexes to efficiently answer SPARQL queries, simplifying the join evaluations. (iii) To quantify the performance impact of using bitmap indexes, we compare our approach to the state-of-the-art triple-store RDF-3X. We find that our bitmap index-based approach to answering queries is up to an order of magnitude faster for a variety of SPARQL queries, on gigascale RDF data sets.

Keywordssemantic data, RDF, SPARQL query optimization, compressed bitmap indexes, large-scale data analysis

1 Introduction

The Resource Description Framework (RDF) was devised by the W3C consortium as part of the grand vision of a semantic web[1]. RDF is now a widely-used standard for representing collections of linked data [4, 5]. It is well-suited for modeling network data such as socio-economic relations and biological networks [27, 30]. It is also very useful for integrating data from dynamic and heterogeneous sources, in cases where defining a schema beforehand might be difficult. Such flexibility is key to its wide use. However, the same flexibility also makes it difficult to answer queries quickly. In this work, we propose a new strategy using bitmap indexes to accelerate query processing.

Each record in the RDF data model is a triple of the form ⟨`subject`, `predicate`, `object`⟩. If these records are stored in a data management system as a three-column table, then all queries except a few trivial ones would require self-joins, and this would be inefficient in practice. To speed up the query answering process, there have been a number of research efforts based on modifying existing data base systems and developing specialized RDF processing systems. For

[1]More information about RDF can be found at http://www.w3.org/RDF/.

example, popular commercial database systems (DBMS) such as ORACLE have added support for RDF [22]. A number of research database management systems have also been applied to RDF data [1, 29]. Special-purpose RDF storage systems include Jena [2] and hyperGraphDB [3,4].

The most commonly used query language on RDF data is called SPARQL [26]. The need to support queries in this language also makes it more necessary to develop specialized RDF processing systems.

The critical piece of technology in these new RDF processing systems and additions to DBMS systems is the indexing technology to enable efficient query answering on RDF data. The key observation underlying our work, and many others in the past, is that RDF data is a table with only three columns, and therefore can only have a small number of combinations for composite and join indexes [18, 23, 33]. More specifically, there are only three single-column indexes, three two-column indexes and three join indexes involving join conditions of the form (in SQL notation) "S.subject = T.subject," "S.subject = T.object," and "S.object = T.object." Given this observation, we can design data structures to implement all these possible combinations to satisfy most common queries, as demonstrated by RDF-3X [23] and RDFKB [21].

Our work takes the above idea and builds a significantly more efficient set of data structures for both storing the RDF data and answering SPARQL queries. By utilizing a compute-efficient bitmap compression technique and carefully engineering the query evaluation procedure, we dramatically reduce the query processing time compared to the state-of-the-art software package RDF-3X. More specifically, our contributions include the following:

- We present an efficient processing strategy for parsing the raw RDF data, building dictionaries of unique entities, and creating compressed bitmap indexes of the data. In addition, we exploit shared-memory parallelism on multicore systems to speed up key routines in this strategy. With our approach, we construct the dictionaries and bitmap indexes for a 500-million record RDF data set (raw data size of 62 GB) in less than 30 minutes on a workstation.

- We utilize the constructed bitmap indexes to accelerate SPARQL query evaluation. We present an outline of our query evaluation strategy in Section 4.

- We conduct an extensive experimental study to gauge the impact of using bitmap indexes. Performance results indicate that our strategy either matches or outperforms the high-performance triple store RDF-3X for a variety of complex RDF queries. Section 6 presents our observations from this study.

2 Related Work

In this section, we briefly review related work on indexing RDF data. Since the RDF records can be treated as either nodes of a graph or rows of a three-column table, we classify the indexing techniques accordingly into graph indexing techniques or table indexing techniques.

Before RDF become widely used, there were already a number of innovative indexing methods for XML data [10, 15]. Since the advent of RDF, many new techniques specifically for RDF data have been developed [12, 14, 16]. Most of these indexing methods are implemented using prototype

[2] http://openjena.org/.

[3] http://www.hypergraphdb.org/.

[4] More specialized RDF processing systems are listed at http://semanticweb.org/wiki/Tools.

systems and cannot easily be integrated into other data processing systems. Overall, we believe the table indexing methods are more likely to be compatible with an existing data processing system and therefore concentrate on such methods in this work.

The most common indexing techniques in database systems are variants of B-Trees [8] or bitmap indexes [25]. The techniques for indexing RDF data generally follow these two prototypical methods as well. For example, Fletcher and Beck proposed a technique called Three-way Triple Tree [9], and McGlothlin and Khan proposed a indexing method using bitmaps [20]. Of course, there are also creative combinations of multiple indexing approaches, for example, Nguyen et al. propose combining hashing with B-trees to produce B+Hash Trees [24]. Typically, a B-Tree based approach is more efficient for highly selective queries, such as those looking for one or a few records. Because most of the query conditions on RDF data selects more than a handful of records, the bitmap indexing approach is generally more efficient. Among the existing indexing methods, RDF-3X is the best among the B-Tree variants [23]; however, two bitmap indexing methods, BitMat and RDFJoin, have demonstrated performance on par with RDF-3X [2, 19]. They also highlight the effectiveness of a bitmap-based approach and motivate us to consider even more optimized bitmap indexes.

The BitMat index creates a 3D bit-cube with the three dimensions being subject, predicate, and object. This cube is compressed and loaded into memory before answering any queries [2, 3]. This technique has been shown to be quite efficient, but due to its reliance on the whole bit-cube to be in memory, it is difficult to scale to larger datasets.

The RDFJoin technique breaks the 3D bit-cube used by BitMat into six separate bit matrices. Each of these bit matrices can be regarded as a separate bitmap index, and therefore can be used independently from other other. Thus, the RDFJoin approach is more flexible and can be applied to larger datasets [20]. In this work, we further enhance the approach by only loading the necessary bitmaps into memory and reducing the memory requirement to the minimum. This not only reduce the I/O costs, but also reduce the amount of time needed for CPU also.

To accelerate join operations, a tempting option for the authors is to use the bitmap cross-product option [17]. Here, the result of the join is directly computed using bitmaps that represent the subjects and objects. It has been shown to be quite efficient and requires no explicit join indexes. However, this strategy is better-suited for tables with many columns where the joins can be performed on arbitrary combinations of columns. For RDF data, there are only three columns and only three commonly used join conditions. Therefore, a more specialized indexing strategy might be more efficient, as demonstrated by McGlothlin and Khan [20, 21]. We adapt a compute-efficient compression method to further improve the bitmap indexes for RDF data.

Most compression methods are designed to minimize the compressed size of the index. However, in the case of bitmap indexes, because the compressed bitmaps are extensively used in bitwise logical operations, it is more important that logical operations can be efficiently performed on the compressed bitmaps directly. The specific compression method we use is called Word-Aligned Hybrid (WAH) code [35]. It is a hybrid between run-length compression and literal representation of raw bits. It is very simple so that bitwise logical operations can directly operate on the compressed bitmaps without producing uncompressed bitmaps. A key feature of this compression is that during bitwise logical operations, it accesses data as whole words instead of individual bytes or bits. This allows the operations to better match the capability of modern CPUs, and further improves its overall efficiency. At the same time, WAH also offers enough compression that the worst-case index size is a linear function of the number of records [35]. In computational complexity theory, this is

regarded as optimal.

Compression techniques have also been used to directly compress the RDF data [1, 29]. This accelerates query processing by reducing the amount of I/O needed to answer queries. In this work, we replace the string values with dictionaries and compressed bitmaps, which represents the RDF records very compactly. This also turns most computations while answering a query into bitwise logical operations, where our compute efficient compression can show its full benefit. At the same time, the bitmaps can provide statistics about the string values needed for query planning, which can further accelerate certain queries.

Since most of the operations on the RDF records are querying instead of updates, the systems for managing RDF data are similar to data warehouses. This has led to a number of researchers to explore the possibility of transplanting On-Line Analytical Processing (OLAP) techniques to the RDF stores [7]. Furthermore, a number of graph summarization techniques such as Trace Equivalence and Bi-similarity have also been proposed in recent literature [13].

3 Bitmap Index Construction

We next explain the data structures used in our work. We describe them as bitmap indexes in this work, because each of them consists of a set of key values and a set of compressed bitmaps, similar to the bitmap indexes used in database systems [25, 35]. However, the key difference is that each bitmap may not necessarily correspond to an RDF record (or a row), as in database systems. For RDF data, one can construct the following sets of bitmap indexes:

Column Indexes. The first set of three bitmap indexes are for three columns of the RDF data. In each of these indexes, the key values are the distinct values of subjects, predicates, or objects, and each bitmap represents which record (i.e., row) the value appears in. This is the standard bitmap index used in existing database systems [25, 35].

Unlike conventional bitmap indexes, our indexes for subject and object share the same dictionary. This strategy is taken from the RDFJoin approach [20]. It eliminates one dictionary from the three bitmap indexes, and allows the self-join operations to be computed using integer keys instead of string keys. This is a trick used implicitly in many RDF systems.

Composite Indexes. We can create three composite indexes, each with two columns as keys. The keys are composite values of predicate-subject, predicate-object, and subject-object. This ordering of the composite values follows the common practice of RDF systems. As in normal bitmap indexes, each composite key is associated with a bitmap. However, unlike the normal bitmap index where a bitmap is used to indicate which rows have the particular combination of values, our bitmap records values the other column has. For example, in a composite index for predicates and subjects, each bitmap represents what values the objects have.

In a normal bitmap index, there are many columns not specified by the index key. Therefore, it is useful for the bitmap to point to rows containing the specified key values, so that any arbitrary combination of columns may be accessed. However, in the RDF data, there are only three columns. If the index key contains information about two of the three columns already, directly encoding the information about the third column in the index removes the need to go back the data table and is a more direct way of constructing an index data structure.

Table 1: Summary of bitmap indexes and dictionaries employed in our RDF query evaluation scheme. In the terms below, n_X (X being P, S, O, or SO) refers to cardinality of column X.

Index	**# bitmaps**	**Size** of each **uncompressed** bitmap
PSIndex	$O(n_P \cdot n_S)$	n_{SO}
POIndex	$O(n_P \cdot n_O)$	n_{SO}
PSIndex (summary)	n_P	n_{SO}
POIndex (summary)	n_P	n_{SO}

Dictionary	**Mapping**	# entries
Predicates	String to integer identifier	n_P
Subject-Object	String to integer identifier	n_{SO}

To effectively encode the values of the third column in a bitmap, we use a bitmap that is as long as the number of distinct values of the column. In the example of a predicate-subject index, each bitmap has as many bits as the number of distinct values in objects. In our case, the bitmap has as many bits as the number of entries in the subject-object dictionary. To make it possible to add new records without regenerating all bitmap indexes, our dictionary assigns a fixed integer to each known string value. A new string value will thus receive the next available integer. When performing bitwise logical operations, we automatically extend the shorter input bitmap with additional 0 bits. This allows us to avoid updating existing bitmaps in an index, which can reduce the amount of work needed to update the indexes when new records are introduced in a RDF data set.

Join Indexes. A normal join index represents a cross-product of two tables based on an equality join condition. Because the selection conditions in SPARQL are always expressed as triples, the join operations also take on some special properties, which we can take advantage of when constructing the join indexes. Note that for SPARQL queries, joins are typically across properties. Thus, the most commonly-used join indexes for RDF data would map two property identifiers to a corresponding bitmap, and there can be three such indexes based on the positions of the variable. In the current version of our RDF processing system, we chose not to use construct join indexes due to the observation that most of the test queries could be solved efficiently with just composite indexes. We will investigate use of join indexes for query answering in future work.

Summary Composite Indexes. In addition to the composite indexes, we also create two other bitmap indexes: summary POIndex and summary PSIndex. The summary POIndex composite index is a collection of n_P bitmaps. For each predicate *pid*, *summary* $PSIndex[pid]$is given by the bit vector $\bigcup_{i=1}^{n_S} PSIndex[pid, sid]$. Thus, this bitmap captures all possible object identifiers with which the predicate *pid* may occur in the data set. Since the number of predicates is typically much lower than the distinct subject-object count, it is not very expensive to maintain these summary composite indexes.

3.1 Parallelization

The salient features of the bitmap indexes and dictionaries that we construct for query evaluation are summarized in Table 1. The dictionary and bitmap index construction steps are computationally very expensive, with the cost increasing super-linearly on increasing data size. Prior work reports that the data ingestion step is extremely slow for popular open-source RDF stores such as Jena and Sesame [28]. We implement all the construction steps in a modular manner, and hence they are amenable to incremental parallelization. In particular, we perform the following steps to construct the dictionaries and composite indexes:

1. Read the RDF data into an in-memory buffer in batches and parse each new line to identify the subject, predicate, and object strings.
2. Use a hash table to determine the unique predicate strings. Assign each of these strings unique integer identifiers and write the dictionary to disk.
3. Sort the triples by subject, iterate through list of subjects removing duplicates, and create a list of unique subject strings.
4. Sort the triples by object, iterate through list removing duplicates, and create a list of unique object strings.
5. Merge the sorted subject and object string arrays, identify list of unique strings, and create the combined subject and object dictionary. Write the dictionary to disk.
6. Replace the subject and object strings with their integer identifiers. Write the integer triple list to disk.
7. Read the integer triples from disk. To create PSIndex, sort the integer triples first by Predicate, and second by Subject. Identify all unique objects for each PS pair, create corresponding compressed bitmap. Write all the created bitmaps to disk.
8. Construct POIndex in an approach similar to the previous step.
9. Create the summary composite indexes.

Observe that we use the sort routine extensively, for finding unique subjects, objects, as well as for composite index creation. This is due to the fact that the number of unique subject-object strings is typically quite large in RDF data sets, about the same order as the number of triples. Sorting mitigates irregular memory references that may be required when maintaining a hash table. We also perform all the sorts in parallel, thus incrementally parallelizing the construction phase without affecting other steps in the construction process.

4 Query Evaluation and Optimization

SPARQL is a query language that expresses conjunctions and disjunctions of triple patterns. Each conjunction, denoted by a dot in SPARQL syntax, nominally corresponds to a join. A SPARQL query can also be viewed as a graph pattern-matching problem: the RDF data represents a directed multigraph, and the query corresponds to a specific pattern in this graph, with the possible degrees

of freedom expressed via wildcards and variables. Figure 1 gives an example SPARQL query (more queries are listed in the Appendix). Informally, this query corresponds to the question "produce a list of all scientists born in a city in Switzerland who have/had a doctoral advisor born in a German city". This query is expressed with six triple patterns, and each triple pattern can either have a variable or a literal in the three possible positions. The goal of the query processor is to determine all possible variable bindings that satisfy the specified triple patterns.

SPARQL Query:

```
select ?p where {
  ?p      <type>             ''scientist'' .
  ?city1  <locatedIn>        ''Switzerland'' .
  ?city2 <locatedIn>         ''Germany'' .
  ?p     <bornInLocation>     ?city1 .
  ?adv   <bornInLocation>     ?city2 .
  ?p     <hasDoctoralAdvisor> ?adv .
}
```

Corresponding Query Graph:

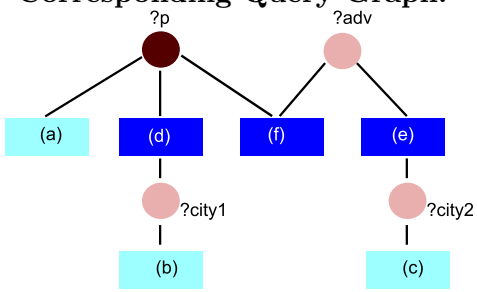

Figure 1: An example SPARQL query and a graph representation of the query triple patterns. The rectangular nodes in the graph represent triple patterns. The labels (a)-(f) correspond to the ordering of the patterns in the query.

To understand how the constructed bitmap indexes can be utilized to answer a SPARQL query, consider the "graph" representation of the query shown in Figure 1. Each triple pattern is shown as a rectangular node. Two triple nodes are connected via query variables they may share, and these variables are represented using circular nodes. Further, the triple patterns are colored based on the number of variable positions in the pattern. The light blue-colored blocks have one variable and one literal in their pattern, whereas the dark blue blocks represent patterns with two variables. Similarly, the dark brown circular node represents the output variable, and the nodes in light blue color are other variables in the query. Such a query graph succinctly captures the query constraints, and forms the basis for a possible query evaluation approach.

For query evaluation, consider representing each variable using a bitmap. For instance, the variable p can be initialized to a bitmap of size n_{SO} (where n_{SO} is the cardinality of the combined subject-object dictionary), with all subject bits set to 1. Observe that triple patterns that have only one variable in them can be resolved by composite index (in our case, PSIndex and POIndex) lookups. For instance, the key corresponding to predicate <*type*> and object "scientist" can be determined using dictionary lookups, and then a bit vector corresponding to all possible subjects that satisfy the particular condition can be obtained with a single composite index lookup. Performing a conjunction just translates to performing a bitmap logical "AND" operation with the initialized bitmap. Similarly, we can initialize and update bitmaps corresponding to *city1* and *city2* in the

figure. The other triples (d), (e), and (f) have two variables in their pattern, and so we are required to perform joins. The bit vectors give us a sorted list of index values, and so we employ the nested loop merge join to determine the final binding.

We can utilize auxiliary information in the bitmap indexes to guide the join ordering. Another useful optimization we implement is to utilize the summary PSIndex and POIndex data structures. The summary indexes give, for a particular predicate, the set of Ll possible subjects or objects that could appear along with them, in the form of a bit vector. We can perform a bitmap logical AND operation of the summary PSIndex vector with the current binding associated with a variable *prior to* performing the join, in order to further reduce the number of set bits in the bitmap. For instance, patterns (d) and (e) in the example have the predicate <*bornInLocation*>. We can thus perform an AND of the corresponding summary POIndex bitmaps with *p* and *adv*, and an AND of the summary PSIndex bitmaps with *city1* and *city2*. Reducing the number of set bits in the bitmaps clearly reduces the computational complexity of the final required nested-loop merge join computation.

The UNION expression is an uncommon feature of SPARQL, which allows certain forms of disjunctions. This feature can be incorporated in our processing by performing bitmap logical "OR" computations.

The current version of our query processing system does not include a SPARQL query parser. Hence we manually parse the query and implement one of the many possible join orders. Note that the intent of this work is to demonstrate the feasibility of bitmap indexes for RDF data processing and understand its strengths and limitations. In future work, we will investigate automated query optimization and advanced join ordering algorithms. We will also adopt a calculus representation suitable for working with bitmap indexes, such as the one proposed by Atre et al. [2] for SPARQL query optimization using the BitMat data representation.

Thus, the key primitives in our query-answering methodology are dictionary lookups, composite index lookups, summary composite index lookups, bit vector AND and OR operations, and finally nested loop merge joins after decompressing bit vectors. We were able to express all the benchmark queries, some with as many 15 triple patterns, compactly using these primitives. The query required less than 20 lines of source code in almost all cases.

We use FastBit software to perform these operations. In most cases, the lookup operations are performed with integer identifiers of the string values obtained through the dictionaries. These integer identifiers are directly used as indices into arrays of bitmaps in the bitmap indexes. FastBit stores bitmaps with Word-Aligned Hybrid (WAH) compression. The bitwise logical operations can be directly performed on the compressed bitmaps. A theoretical analysis has shown that the bitwise AND and bitwise OR operations on WAH compressed bitmaps scale linearly with the compressed sizes of these bitmaps, which is optimal in computational complexity [35]. In a number of empirical studies, operations on WAH compressed bitmaps are also found to much more efficient than on other compressed bitmaps [34].

5 Experimental Evaluation

5.1 Data sets

We choose a variety of data sets and test instances to test our new FastBit-based query evaluation scheme. First, we experiment with synthetic data sets of different sizes using the Lehigh University

Benchmark suite LUBM [11]. LUBM is a popular benchmark for evaluating triple stores, with a recommended list of fourteen queries that stress different aspects related to query optimizations. The LUBM queries are listed in the Appendix. The number of predicates generated in this data set is 18.

We also present query results with the Yago [31] dataset, which is comprised of facts extracted from Wikipedia. This dataset contains about 40 million triples, and the number of distinct predicates (337K) is significantly higher than LUBM. We use a set of queries that were previously used to evaluate RDF-3X.

Further, we use large subsets of two datasets: the Billion Triples Challenge [6] data and the UniProt [32] collection. The Billion Triples dataset encapsulates public domain web crawl information, whereas UniProt is a repository for protein-related information. We implement three sample queries for each of these datasets. Both these datasets are significantly more complex, noisy, and heterogeneous compared to LUBM. We use queries recommended by the UniProt publishers, and ones similar to prior RDF-3X query instances.

5.2 Choice of Competing Approaches

There are numerous production and prototype research triple-stores available for comparison, a majority of which are freely available online[5]. In this paper, we chose to compare our bitmap index strategies against RDF-3X [23]. We use version 0.3.6 (the latest version) for most of our experiments (unless otherwise specified), and version 0.3.5 for a few runs. RDF-3X is a production-quality RDF-store widely used by the research community for performance studies, and prior work shows that it is significantly faster, sometimes by up to two orders of magnitude, than alternatives such as MonetDB and Jena-TDB. We also experimented with the bitmap indexing-based software BitMat [2], but found that RDF-3X consistently outperforms BitMat for a variety of queries. In particular, the new version v0.3.6 of RDF-3X presumably enables some optimizations that fix a previous shortcoming when processing high selectivity complex queries.

The RDF-3X SPARQL query processing routine includes a parser and performs selectivity estimation and a dynamic programming-based query optimization phase before the actual operator tree execution to realize the final output. For a fair comparison with our work, we only time the operator tree execution phase when presenting RDF-3X results.

5.3 Test Systems and Software

The primary test machine data5 is a typical Linux workstation with a quad-core Intel Xeon processor with a clock speed of 2.67GHz, 8MB L2 cache, and 8GB RAM. The disk system used to store the test data is a software RAID concatenating two 1TB SATA disks in RAID0 configuration. We also present results on a 256 GB SSD drive on the same system. The second test machine named euclid is a shared resource at NERSC [6]. It is a Sunfire x4640 SMP with eight 6-core Opteron 2.6 GHz processors and 512 GB of shared memory. On this system, the test files are stored on a GPFS file system shared by thousands of users. Therefore, we may expect more fluctuations in I/O system performance.

We use FastBit v1.2.2 for implementing our bitmap index-based RDF data processing approach. We built the codes using the GNU C++ compiler v4.4.3 on data5 and the PGI C++ compiler v10.8

[5]Please see http://semanticweb.org/wiki/Tools for a list of tools.

[6]More information about NERSC and euclid can be found at http://www.nersc.gov.

Table 2: Data, Index, and Database sizes for different data sets.

Data set # triples	**LUBM 1M**	**LUBM 50M**	**LUBM 500M**	**Yago 40M**	**UniProt 220M**	**BTC 626M**
Raw data	0.125	6.27	62.30	3.56	30.58	65.19
FastBit Dictionaries	0.032	0.79	8.22	1.30	3.05	2.48
FastBit Indexes	0.016	1.59	15.41	1.20	6.30	15.03
RDF-3X DB	0.058	2.83	33.84	2.75	—	—

on euclid. For parsing the data, we use the Raptor RDF parser utility (v2.0.0).

For all the queries, we present cold cache performance results, which correspond to the first run of the query, as well as "warm cache" numbers, which are an average of ten consecutive runs, excluding the first.

6 Results and Discussion

6.1 Index construction and sizes

Table 2 lists the sizes of the FastBit dictionaries and indexes after the construction phase. We observe that the cumulative sum of the dictionary and index sizes is substantially lower than the raw data size for all the data sets. As a point of comparison, we present the size of the RDF-3X B-tree indexes (which internally stores six compressed replicas of the triples compactly) for each of these datasets. We could not construct indexes for the larger data sets using RDF-3X v0.3.6 due to time constraints (the latest version was released early February). However, for the data sets studied, our approach uses slightly lower disk space than RDF-3X.

The dictionary and index construction times range from 20 seconds on data5 for the 1M triple LUBM data set, to nearly four hours for the BTC 626M triple data set on euclid. Figure 2 gives the overall parallel performance (on eight cores of euclid) and the performance breakdown of the various steps involved in the construction phase. The overall parallel speedup is 2.66×, and the serial reads and writes to GPFS become a significant bottleneck in the parallel run. Among the other steps, the combined subject-object dictionary construction takes the maximum time, as it involves two sorts of variable-sized strings.

6.2 LUBM Query Performance

We next evaluate the performance of our bitmap index based approach for LUBM data sets of three different sizes.

Comparison to RDF-3X. In Table 3, we compare the cold and warm caches performance of queries for the LUBM-1M and the LUBM-50M data sets. We do not observe a substantial difference between the cold and warm cache times (the difference is not as pronounced as RDF-3X), which may indicate that the indexes may be already cached in main memory and I/O activity is minimal. Overall, we observe that our strategy outperforms RDF-3X by a significant margin in the warm cache case, particularly for the 1M dataset. Studying the queries individually, we observe that the speedup is higher for simple two or three triple pattern queries (such as queries 1, 5, 10, and 14). The results for the slightly more complex queries (queries 2, 8, 9) are mixed: RDF-3X is faster for

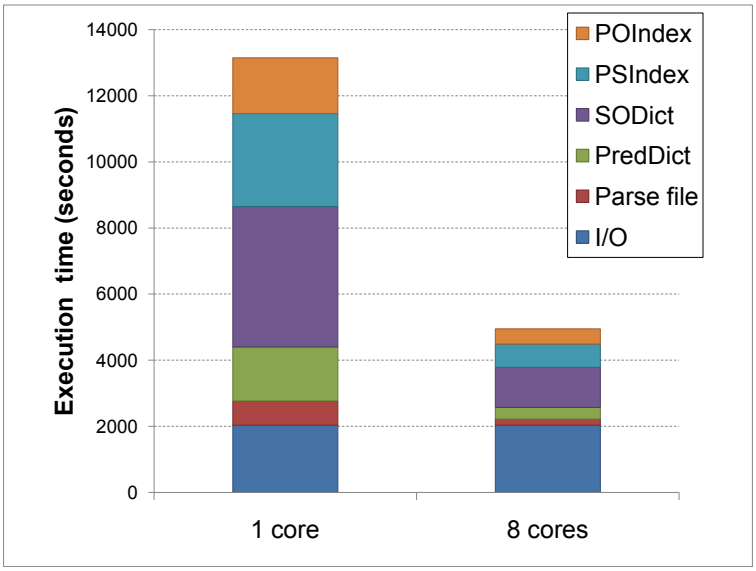

Figure 2: Serial and parallel (on eight cores of euclid) execution times for various steps in the bitmap index and dictionary construction phase for the Billion Triples data set. The overall parallel speedup achieved is 2.66×.

query 2 for both the 1M and the 50M data sets, whereas our bitmap index-based approach is faster for query 8 on both the data sets. We surmise that this may be because we picked a non-optimal join ordering when executing query 2. Table 4 presents performance results for the same set of queries on a 500M data set, but on the euclid system. We also ran an older version of RDF-3X on this system. Interestingly, query 2 is significantly slower and our test harness times out for this particular instance. Thus, the new version of RDF-3X presumably includes some optimizations to handle these tough queries. Another trend apparent on investigating the relative performance is that the overall average speedup reduces as the data size increases.

We also separately analyzed Queries 2, 6, 9, and 14, as their execution times were significantly higher than the rest of the queries. We wanted to know whether this was due to join computations (performing extraneous work) or dictionary lookups (potentially expensive random accesses). Table 5 gives the query execution times and the output triple sizes for these four queries. In addition, we defer materializing the output in order to isolate the time incurred in dictionary lookups. We see that the relatively-simpler queries 6 and 14 achieve a substantial speedup when the lookups are not timed. This is expected, as the query selectivity is very low. Similarly, the execution times of the relatively high selectivity queries 2 and 9 is dominated by bitmap and join computations.

Query execution time scaling with data size. We next study the query execution time scaling as a function of the data size. On comparing the execution times for queries as the data size is increased from 1M to 500M, we see that the times, for most queries, increase sub-linearly for both RDF-3X and our approach. The execution times of low selectivity queries such as Q14 shoot up as the data size increases, but that is mostly because of materializing the final result.

Impact of Flash memory. We performed similar experiments with data residing on the higher bandwidth and lower latency SSD drive on the data5 system. The query execution times for a representative data size, LUBM-50M, are tabulated in Table 6. Interestingly, we observe that the cold

Table 3: LUBM benchmark SPARQL query evaluation times (in milliseconds) for data sets of different sizes on data5-sata.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
LUBM-1M, Cold caches							
FastBit	0.078	16.2	0.098	0.150	0.118	2.85	0.12
RDF-3X	15.5	32.3	7.3	1.27	1.14	92.4	1.25
LUBM-1M, Warm caches							
FastBit	0.008	15.7	0.026	0.042	0.028	2.59	0.032
RDF-3X	0.385	10.2	0.385	0.553	1.11	76.1	0.89
Speedup	48.1×	0.65×	14.8×	13.16×	39.6×	29.4×	27.8×
LUBM-50M, Cold caches							
FastBit	0.30	1320	1.26	0.65	0.34	139	0.643
RDF-3X	0.43	572	2.9	0.75	2.1	4150	4.62
LUBM-50M, Warm caches							
FastBit	0.167	1311	0.92	0.40	0.19	135	0.46
RDF-3X	0.31	544	0.193	0.70	1.95	4021	1.52
Speedup	1.86×	0.42×	0.21×	1.75×	10.26×	29.8×	3.30×

	Q8	Q9	Q10	Q11	Q12	Q13	Q14
LUBM-1M, Cold caches							
FastBit	5.41	11.2	0.08	0.07	0.215	0.076	7.94
RDF-3X	63.8	46.6	1.04	0.95	3.45	0.414	256
LUBM-1M, Warm caches							
FastBit	4.67	10.2	0.016	0.11	0.104	0.011	7.91
RDF-3X	50.6	23.3	0.32	0.38	2.27	0.187	244
Speedup	10.83×	2.23×	20×	3.45×	21.8×	17×	30.85×
LUBM-50M, Cold caches							
FastBit	7.85	9457	0.313	0.263	2.61	0.36	636
RDF-3X	55.6	1431	1.65	0.41	17.2	3.9	14190
LUBM-50M, Warm caches							
FastBit	6.34	9288	0.179	0.148	2.34	0.34	467
RDF-3X	50.4	1369	0.336	0.35	7.44	1.7	13770
Speedup	7.95×	0.15×	1.87×	2.36×	3.17×	5.0×	29.5×

and warm cache numbers are identical to the SATA case. This is expected, as the dictionaries and indexes fit in main memory for this problem size and thus the overall I/O effects are insignificant.

Table 4: LUBM benchmark SPARQL query evaluation times (in milliseconds) for a 500 million triple data set on euclid.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
LUBM-500M, Cold caches							
FastBit	1.92	17481	17.2	3.2	0.62	2560	1.75
RDF-3X (v0.3.5)	1.58	—	85.9	199.6	1.25	91300	560.6
LUBM-500M, Warm caches							
FastBit	1.73	8344	7.81	1.21	0.41	2344	1.21
RDF-3X (v0.3.5)	0.875	—	0.984	2.344	1.11	80039	3.47
Speedup	0.51×	—	0.13×	2.71×	2.68×	34.1×	2.87×

	Q8	Q9	Q10	Q11	Q12	Q13	Q14
LUBM-500M, Cold caches							
FastBit	9.43	278.1	1.27	2.44	15.7	5.32	11231
RDF-3X	204.2	41.2	870.1	30.2	1051.12	15082.3	—
LUBM-500M, Warm caches							
FastBit	7.23	140.4	0.38	1.52	11.51	2.53	10682
RDF-3X	71.8	28.1	1.01	0.94	124.2	18.5	—
Speedup	9.93×	0.2×	2.66×	0.62×	10.79×	7.31×	—

Table 5: Impact of deferred dictionary lookups when materializing final result: LUBM benchmark SPARQL query evaluation times (in milliseconds) for a 50 million triple data set on data5-sata.

	Q2	Q6	Q9	Q14
Warm caches time	1311	135	9288	467
Time with deferred dictionary lookups	1317	13.2	8395	51.5
Speedup with optimization	0.99×	10.23×	1.11×	9.07×
Output size	2754	911982	47655	2864582

Table 6: LUBM benchmark SPARQL query evaluation times (in milliseconds) for a 50 million triple data set on data5-ssd.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Cold caches time	0.292	1340	1.25	2.02	0.33	141	0.66
Warm caches time	0.156	1329	0.897	0.38	0.18	132	0.44

	Q8	Q9	Q10	Q11	Q12	Q13	Q14
Cold caches time	7.61	9510	0.30	0.259	2.32	0.356	652.3
Warm caches time	6.19	9261	0.17	0.14	2.05	0.343	451.5

Table 7: YAGO data set SPARQL query evaluation times (in milliseconds) on data5-sata.

	A1	**A2**	**B1**	**B2**	**C1**	**C2**
Cold caches						
FastBit	11.4	13.67	241.8	3.55	1145	58.3
RDF-3X	15.5	18.9	64.9	31.7	1668	44.8
Warm caches						
FastBit	2.13	4.2	205.2	1.64	964.5	35.1
RDF-3X	7.5	7.8	53.9	21.1	271.2	38.2
Speedup	3.52×	1.86×	0.27×	12.87×	0.28×	1.09×

Table 8: UniProt and Billion Triple datasets SPARQL query evaluation times (in milliseconds) on euclid.

	UniProt			Billion Triples		
	Q1	**Q2**	**Q3**	**Q1**	**Q2**	**Q3**
Warm caches time	1.71	262	30.4	12.35	443.42	378.21

6.3 Performance on Multi-pattern Complex SPARQL Queries

We next discuss results for three other fixed-size datasets. Table 7 gives the FastBit and RDF-3X performance achieved for queries on the Yago data set. The queries are classified into three types: type A consisting of oriented facts, type B that are relationship oriented, and type C examining relationships with unknown predicates. One common characteristic of all these queries is that they are all highly selective, leading to result sets with less than 100 triples in each case. RDF-3X outperforms our approach for queries B1 and C1. These are queries using the SPARQL distinct keyword, which requires additional computation in case of bitmap indexes. We will investigate optimizing these queries in future work. For the rest of the queries, FastBit outperforms RDF-3X by a substantial margin, both for the warm and the cold cache cases.

In Table 8, we summarize performance achieved for sample queries on the large-scale UniProt and Billion Triple data sets. The summary POIndex and PSIndexes are very useful in case of the UniProt queries, where there are several triple patterns sharing the same join variable. These indexes help prune the tuple space significantly, and the query execution times are comparable to previously-reported RDF-3X numbers. We could not build the RDF-3X database for these datasets, as we encountered a parsing problem. We will resolve this issue in future work.

Table 9 summarizes the overall performance improvement achieved for queries using FastBit versus RDF-3X, when taking the geometric mean of the execution times into consideration. We observe that we outperform RDF-3X for both the LUBM and the Yago datasets.

7 Conclusions and Future Work

This paper presents the novel use of compressed bitmaps to accelerate SPARQL queries on large-scale RDF repositories. Our experiments show that we can process queries with as many as 10 to

Table 9: FastBit query evaluation performance speedup achieved (taking the geometric mean of the execution times across queries) over RDF-3X for various data sets. † Speedup w.r.t. RDF-3X v0.3.5 on euclid.

	LUBM-5M	**LUBM-50M**	**LUBM-500M†**	**YAGO**
Speedup	12.96×	2.62×	2.81×	1.38×

15 triple patterns, and query execution times compare very favorably to the current state-of-the-art results. Bitmap indexes are space-efficient, and we claim that bitvector operations provide an intuitive and convenient mechanism for expressing and solving ad-hoc queries. The set union and intersection operations that are extensively used in SPARQL query processing are extremely fast when mapped to bitvector operations.

We plan to extend and optimize our RDF data processing system in future work. First, we will speed up data ingestion even further by exploiting parallel I/O capabilities and distributed memory parallelization of the sort steps. Our current dictionary and index creation schemes provision for incremental updates to the data. We intend to study the cost of updates, both fine-grained as well as bulk loads.

We do not support a full SPARQL query parser and automated query optimization with guided join ordering in our current implementation; we plan to research these problems in future work. It is currently quite cumbersome to express path-based queries in SPARQL, and our indexes have not been designed with such queries in mind. However, we note that the composite indexes provide a compressed representation of the implicit RDF graph, and we can utilize these indexes for developing parallel graph algorithms for traversal, shortest paths, and connected components.

Acknowledgments

This work is supported by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[2] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In *WWW*, pages 41–50, 2010.

[3] Medha Atre, Jagannathan Srinivasan, and James A. Hendler. Bitmat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In *International Semantic Web Conference (Posters & Demos)*, 2008.

[4] Tim Berners-Lee. Linked data. *International Journal on Semantic Web and Information Systems*, 4:1, 2006.

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

[6] Semantic web challenge. http://challenge.semanticweb.org/, last accessed Feb 2011.

[7] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip Yu. Graph OLAP: a multi-dimensional framework for graph data analysis. *Knowledge and Information Systems*, 21:41–63, 2009.

[8] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[9] George H. L. Fletcher and Peter W. Beck. A role-free approach to indexing large RDF data sets in secondary memory for efficient SPARQL evaluation, November 07 2008.

[10] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB '97*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[11] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semant.*, 3:158–182, October 2005.

[12] Andreas Harth and Stefan Decker. Optimized index structures for querying RDF from the web. In *LA-WEB*, pages 71–80. IEEE Computer Society, 2005.

[13] A. Hawash, A. Deik, B. Farraj, and M. Jarrar. Towards query optimization for the data web: disk-based algorithms: trace equivalence and bisimilarity. In *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications*, ISWSA '10, pages 17:1–17:7, New York, NY, USA, 2010. ACM.

[14] Mustafa Jarrar and Marios D. Dikaiakos. Querying the data web: The mashQL approach. *IEEE Internet Computing*, 14(3):58–67, 2010.

[15] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-tree: Indexing XML data for efficient structural joins. In *ICDE*, pages 253–263. IEEE Computer Society, 2003.

[16] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering indexes for branching path queries. In *SIGMOD '02*, pages 133–144, New York, NY, USA, 2002. ACM.

[17] Kamesh Madduri and Kesheng Wu. Efficient joins with compressed bitmap indexes. In *CIKM*, pages 1017–1026, 2009.

[18] James P. McGlothlin. Framework and schema for semantic web knowledge bases. In *AAAI*, 2010.

[19] James P. McGlothlin and Latifur Khan. Efficient RDF data management including provenance and uncertainty. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, IDEAS '10, pages 193–198, New York, NY, USA, 2010. ACM.

[20] James P. McGlothlin and Latifur R. Khan. RDFJoin: A scalable data model for persistence and efficient querying of RDF datasets. Technical Report UTDCS-08-09, Univ. of Texas at Dallas, 2008.

[21] James P. McGlothlin and Latifur R. Khan. RDFKB: efficient support for RDF inference queries and knowledge management. In *IDEAS*, pages 259–266, 2009.

[22] Chuck Murray. RDF data model in oracle. Technical Report B19307-01, Oracle, 2005.

[23] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1:647–659, August 2008.

[24] Minh Khoa Nguyen, Cosmin Basca, and Abraham Bernstein. B+Hash tree: Optimizing query execution times for on-disk semantic web data structures. In Achille Fokoue, Thorsten Liebig, and Yuanbo Guo, editors, *SSWS2010*, pages 96–111, November 2010.

[25] P. O'Neil. Model 204 architecture and performance. In *Proc. HPTS*, volume 359 of *Lecture Notes in Computer Science*, pages 40–59, September 1987.

[26] Eric Prud'Hommeaux and Andy Seaborne. SPARQL query language for RDF. World Wide Web Consortium, Recommendation REC-rdf-sparql-query-20080115, January 2008.

[27] Nicole Redaschi and UniProt Consortium. Uniprot in RDF: Tackling data integration and distributed annotation with the semantic web. Nature Proceedings, 2009.

[28] Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder, and John Sumner. An evaluation of triple-store technologies for large data stores. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, volume 4806 of *Lecture Notes in Computer Science*, pages 1105–1114. Springer Berlin / Heidelberg, 2007.

[29] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *Proc. VLDB Endow.*, 1:1553–1563, August 2008.

[30] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW '07*, pages 697–706, New York, NY, USA, 2007. ACM.

[31] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A large ontology from Wikipedia and WordNet. *Web Semant.*, 6:203–217, September 2008.

[32] Uniprot RDF distribution. ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf/, last accessed Feb 2011.

[33] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1:1008–1019, August 2008.

[34] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, pages 24–35, 2004.

[35] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM TODS*, 31(1):1–38, 2006.

A SPARQL Queries

We list here all the benchmark SPARQL queries used in our bitmap index evaluation.

A.1 LUBM

Most of the suggested LUBM queries require inference and references to the ontology, and so we simplified the queries to avoid inference. The queries that require inference can also be expressed using the SPARQL UNION operator in some cases.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix ub: <lubm-bench.owl#>

Q1:
select ?x where {
  ?x rdf:type ub:GraduateStudent .
  ?x ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0>
}

Q2:
select ?x ?y ?z where {
  ?x rdf:type ub:GraduateStudent .
  ?y rdf:type ub:University .
  ?z rdf:type ub:Department .
  ?x ub:memberOf ?z .
  ?z ub:subOrganizationOf ?y .
  ?x ub:undergraduateDegreeFrom ?y .
}

Q3:
select ?x where {
  ?x rdf:type ub:Publication .
  ?x ub:publicationAuthor <http://www.Department0.University0.edu/AssistantProfessor0> .
}

Q4:
select ?x ?y1 ?y2 ?y3 where {
  ?x rdf:type ub:AssociateProfessor .
  ?x ub:worksFor <http://www.Department0.University0.edu> .
  ?x ub:name ?y1 .
  ?x ub:emailAddress ?y2 .
  ?x ub:telephone ?y3 .
}

Q5:
select ?x where {
  ?x rdf:type ub:GraduateStudent .
  ?x ub:memberOf <http://www.Department0.University0.edu> .
}

Q6:
```

```
select ?x where {
  ?x rdf:type ub:GraduateStudent .
}

Q7:
select ?x ?y where {
  ?x rdf:type ub:UndergraduateStudent .
  ?y rdf:type ub:Course .
  ?x ub:takesCourse ?y .
  <http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?y .
}

Q8:
select ?x ?y ?z where {
  ?y ub:subOrganizationOf <http://www.University0.edu> .
  ?y rdf:type ub:Department .
  ?x ub:memberOf ?y .
  ?x rdf:type ub:UndergraduateStudent .
  ?x ub:emailAddress ?z .
}

Q9:
select ?x ?y ?z where {
  ?x rdf:type ub:UndergraduateStudent .
  ?y rdf:type ub:FullProfessor .
  ?z rdf:type ub:Course .
  ?x ub:advisor ?y .
  ?x ub:takesCourse ?z .
  ?y ub:teacherOf ?z .
}

Q10:
select ?x where {
  ?x rdf:type ub:UndergraduateStudent .
  ?x ub:takesCourse <http://www.Department0.University0.edu/Course0> .
}

Q11:
select ?x where {
  ?x rdf:type ub:ResearchGroup .
  ?x ub:subOrganizationOf <http://www.Department0.University0.edu> .
}

Q12:
select ?x ?y where {
  ?x rdf:type ub:FullProfessor .
  ?y rdf:type ub:Department .
  ?x ub:worksFor ?y .
  ?y ub:subOrganizationOf <http://www.University0.edu> .
}
```

```
Q13:
select ?x where {
  ?x rdf:type ub:FullProfessor .
  ?x ub:mastersDegreeFrom <http://www.University236.edu> .
}

Q14:
select ?x where {
  ?x rdf:type ub:UndergraduateStudent .
}
```

A.2 YAGO

```
A1:
select ?gn ?fn where {
  ?gn <givenNameOf>  ?p .      ?fn <familyNameOf> ?p .
  ?p <type> ‘‘scientist’’ .    ?p  <bornInLocation> ?city .
  ?p <hasDoctoralAdvisor> ?a . ?a  <bornInLocation> ?city2 .
  ?city <locatedIn> ‘‘Switzerland’’ .
  ?city2 <locatedIn>  ‘‘Germany’’ .
}

A2:
select ?n where {
  ?a <isCalled> ?n .    ?a <type> ‘‘actor’’ .
  ?a <livesIn>  ?city . ?a <actedIn> ?m1 .
  ?a <directed> ?m2 .
  ?city <locatedIn> ?s . ?s <locatedIn> ‘‘United_States’’ .
  ?m1 <type> ‘‘movie’’ . ?m1 <producedInCountry> ‘‘Germany’’ .
  ?m2 <type> ‘‘movie’’ . ?m2 <producedInCountry> ‘‘Canada’’ .
}

B1:
select distinct ?n1 ?n2 where {
  ?a1 <isCalled> ?n1 . ?a1 <livesIn> ?c1 . ?a1 <actedIn> ?movie .
  ?a2 <isCalled> ?n2 . ?a2 <livesIn> ?c2 . ?a2 <actedIn> ?movie .
  ?c1 <locatedIn> ‘‘England’’ . ?c2 <locatedIn> ‘‘England’’ .
  filter (?a1 != ?a2)
}

B2:
select ?n1 ?n2 where {
  ?p1 <isCalled> ?n1 . ?p1 <bornInLocation> ?city . ?p1 <isMarriedTo> ?p2 .
  ?p2 <isCalled> ?n2 . ?p2 <bornInLocation> ?city .
}

C1:
select distinct ?n1 ?n2 where {
  ?n1 <familyNameOf> ?p1 .   ?n2 <familyNameOf> ?p2 .
  ?p1 <type> ‘‘Scientist’’ . ?p1 [] ?city .
  ?p2 <type> ‘‘scientist’’ . ?p2 [] ?city .
```

```
  ?city <type> <site> .
  filter (?p1 != ?p2)
}

C2:
select distinct ?n where {
  ?p <isCalled> ?n . ?p [] ?c1 . ?p [] ?c2 .
  ?c1 <type> ‘‘village’’ . ?c1 <isCalled> ‘‘London’’ .
  ?c2 <type> ‘‘site’’ .    ?c2 <isCalled> ‘‘Paris’’ .
}
```

A.3 Billion Triples Dataset

```
prefix geo: <http://www.geonames.org/>,
pos: <http://www.w3.org/2003/01/geo/wgs84 pos#>,
dbpedia: <http://dbpedia.org/property/>,
dbpediares: <http://dbpedia.org/resource/>,
owl: <http://www.w3.org/2002/07/owl#>

Q1:
select ?lat ?long where {
  ?a [] "Eiffel Tower" . ?a geo:ontology#inCountry geo:countries/#FR .
  ?a pos:lat ?lat .      ?a pos:long ?long .
}

Q2:
select ?t ?lat ?long where {
  ?a dbpedia:wikilink dbpediares:List of World Heritage Sites in Europe .
  ?a dbpedia:title ?t .
  ?a pos:lat ?lat .
  ?a pos:long ?long .
  ?a dbpedia:wikilink dbpediares:Middle Ages .
}

Q3:
select ?a ?y where {
  ?a a <http://dbpedia.org/class/yago/Politician110451263> .
  ?a dbpedia:years ?y .
  ?a <http://xmlns.com/foaf/0.1/name> ?n .
  ?b [] ?n .
  ?b <http://purl.org/dc/elements/1.1/subject> ‘‘Blackwater’’ .
}
```

A.4 UniProt

```
prefix uni: <http://purl.uniprot.org/core/>,
uniprot: <http://purl.uniprot.org/>,
schema: <http://www.w3.org/2000/01/rdf-schema#>,
file: <file:///uniprot.rdf#>

Q1:
```

```
select ?protein ?name where {
  ?protein a uni:Protein .
  ?protein uni:encodedBy [ uni:name "CRB" ] .
  ?protein uni:name ?name .
}

Q2:
select ?a ?vo where {
  ?a uni:mnemonic ?vo .
  ?a uni:replacedBy uniprot:uniprot/P62965 .
  ?a a uni:Protein .
  ?a uni:modified ‘‘1990-11-01’’ .
  ?a uni:replacedBy uniprot:uniprot/P62966 .
  ?b uni:modified ‘‘2005-08-30’’ .
  ?b uni:replacedBy uniprot:uniprot/P62964 .
  ?b uni:reviewed ‘‘false’’ . ?b uni:obsolete ‘‘true’’.
  ?b a uni:Protein .
  ?a uni:replacedBy ?ab .
  ?ab [] ?b . ?ab uni:classifiedWith uniprot:keywords/845 .
}

Q3:
select ?a ?vo where {
  ?a schema:seeAlso ?vo .
  ?a uni:annotation file:7A64A6 .
  ?a uni:classifiedWith uniprot:keywords/67 .
  ?a uni:annotation file:7A649B .
  ?a uni:annotation file:7A64AF .
  ?a schema:seeAlso uniprot:embl-cds/AAN81952.1 .
  ?b uni:reviewed "true" . ?b schema:seeAlso uniprot:geneid/1025922 .
  ?b schema:seeAlso uniprot:smr/P0A7A1 .
  ?b schema:seeAlso uniprot:embl-cds/AAP18215.1 .
  ?a uni:replaces ?ab . ?ab uni:replacedBy ?b .
}
```