RDFVector: An Efficient and Scalable Schema for Semantic Web Knowledge Bases

James P. McGlothlin

The University of Texas at Dallas
Richardson, TX, USA
jpmcglothlin@utdallas.edu

1 Problem and Motivation

As the semantic web grows in popularity and enters the mainstream of computer technology, RDF(Resource Description Framework) datasets are becoming larger and more complex. As datasets grow larger and more datasets are linked together, scalability becomes more important. As more complex ontologies are developed, there is growing need for efficient queries that handle inference. In areas such as research, it is vital to be able to perform queries that retrieve not just facts but also inferred knowledge and uncertain information.

Currently, scalability and performance issues limit the semantic web from reaching its full potential. The primary goal of RDFVector is to improve the performance of queries against large RDF datasets, including queries that involve inferred knowledge and probabilistic reasoning. RDFVector is a new persistent data model using bit vectors that is highly efficient. This design allows us to store inferred knowledge with no penalty. Queries become simpler and more efficient.

2 Proposed Approach

Figure 1 provides a high level diagram of our architecture. Double boxes indicate tables. The process flow is clearly indicated by the directional arrows. The solid arrows indicate actions that can change data, and the dashed arrows query data.

The add and delete actions operate against the triples table. The triples table calls the inference engine to add or delete inferred triples. After determining triples to add or delete, the inference engine calls the triples table to perform the action. This cycle demonstrates the recursive nature of our inference solution.

One can see from this diagram that three tables provide the core RDFKB schema: the triples table, PSTable and POTable. There is a single one way arrow connecting them showing that all updates to the bit vectors are triggered by the triples table. The query engine supports the user's queries and utilizes the bit vector tables to execute the queries efficiently.

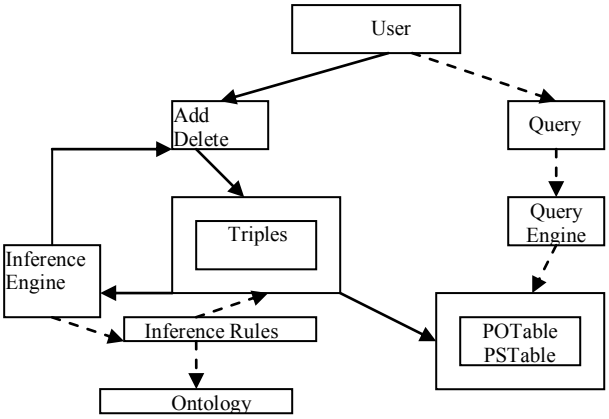

Fig. 1. Architectural Overview of RDFVector

At this point we would like to define our bit vector tables in more detail. The POTable includes four columns: Property, Object, SubjectBitBector and BitCount. The SubjectBitVector has a 1 representing every subject that appears with this property and object in a triple in the dataset. Each URI is dictionary-encoded to a unique identification number that represent the index into the bit vectors. For example, all subjects matching *<?s type text>* can be retrieved from a single tuple in the POTable and returned as a single bit vector, indexed by the identification numbers. Joins can be performed as *and* operations against these bit vectors. The BitCount is simply the number of *on* bits in the vector. We use this information to determine selectivity factors during query optimization. It also important to note that we keep the vector compressed. As the vectors are quite sparse, this substantially reduces memory and storage requirements. Similarily, the PSTable includes property, subject, ObjectBitVector, and bitCount.

Our design is to execute inference at addition time rather than query time. Consider the triple: *<Professor0 type AssociateProfessor>*. The LUBM ontology file[5] will allow us to infer four additional triples:*<Professor0 type Professor>, <Professor0 type Faculty>, <Professor0 type Employee>, <Professor0 type Person>*. Our strategy is to store all inferred triples in the database. Inference rules register with the system and are called when triples are added, so that inferred knowledge can be added as well. Thus, all inferred triples are stored in the database and available to queries. There are 21 subclasses of *Person* in the LUBM ontology. Thus, to query all persons using a schema such as vertical partitioning or RDF-3X requires 21 subqueries and 20 unions. Our solution is both simpler and more efficient; we can query all persons by retrieving a single type from the POTable. No unions or subqueries are required. Most database schemata pay a query performance penalty based on the size of the dataset. Because RDFVector already includes bits in the vectors for every possible combination of known subject, objects, and properties, there is no such penalty.

3 Evaluation Methodology

The goal of RDFVector is efficiency and scalability. Therefore, we evaluate our solution by testing the performance of queries against standardized datasets, and we compare this performance to the current state of the art solutions. We test using hot and cold runs, and, for comparison, we utilize source code from the existing research projects or accepted evaluation research such as [3].

We also execute experiments to test tradeoffs including storage space and memory consumption. For additions and deletions, we use cross validation to select and test subsets of the dataset, and we measure the time to add or delete the selected triples.

4 Query Evaluation and Performance Results

We have implemented RDFVector using the MonetDB[7] database and tested it on a single machine. Our current implementation includes support for inference, for additions and for deletions. We implemented and evaluated experiments with two benchmarks: the Lehigh University Benchmark(LUBM) [5] and the Barton dataset[6]. All of our tests were performed on a Dell M6400 laptop with 8GB RAM.

The LUBM dataset is a synthetic dataset, which allowed us to vary the number of triples in the dataset from 1 to 44 million triples to test scalability. We tested 11 LUBM queries against vertical partitioning and triple store solutions. RDFVector achieved the best performance in all 11 test cases. The performance improvement grew as the dataset size was increased, demonstrating scalability. RDFVector improved performance by over 80.0% in every query for the 44 million triples dataset.

The Barton dataset is an RDF representation of the MIT library catalog. We tested with all 12 queries defined in [3], and compared our results with triple store (PSO and SPO ordered), vertical partitioning[1] and RDF-3X[4]. RDFVector achieved the best performance for all 12 queries during cold runs and for 11 out of 12 queries during hot runs. Table 1 summarizes our results for the Barton dataset.

Table 1. Average query times for Barton dataset

	RDFVector	RDF-3X	VP	PSO	SPO
Average(hot)	*1.02*	1.76	3.53	3.28	4.12
Geo. Mean(hot)	*0.72*	0.97	2.01	2.21	3.21
Average(cold)	*1.54*	4.10	4.68	4.51	5.68
Geo. Mean(cold)	*1.08*	3.24	3.41	3.67	5.07

9 of the 11 LUBM test cases and 3 of the 12 Barton Dataset queries involve inferencing, and RDFVector returned the greatest performance improvement in these tests. In all of these instances, we implemented inference in the other solution by backward chaining. However, we also tried materializing inference in RDF-3X (the next fastest solution) and this actually reduced its performance. This shows that a triple store pays a query performance penalty for increasing the dataset to add inferred triples, and RDFVector does not. Furthermore, RDFVector was also faster in the tests not involving inference.

We also tested the four trade-offs we were able to identify: storage space, memory consumption, time to add triples, time to delete triples. Using vector compression, the size of the Barton Dataset was 3.4GB. The maximum memory consumption was 3.54G to perform all the tests in this paper. The time to add 1 million triples to LUBM was 71 seconds, and the time to delete 1 million triples was 26.7 seconds.

5 Related Work

In this section, we examine related research and products that provide RDF storage and querying.

Vertical partitioning[1] is a schema using column store relational databases. Vertical partitioning creates a two column table for each property, sorted by subject. Vertical partitioning is similar to the PSTable, and provides subject-subject merge joins.

RDF-3X[4] uses a triple store schema combined with indexes, histograms and query optimization. RDF-3X uses its own persistence mechanism, and does not materialize inference.

Hexastore[2] is a main memory sextuple indexing structure for RDF datasets. RDFKB implements 2 of these indices with our bit vector tables. By providing secondary indexing on each table and the triples table, we emulate the other four indices.

Jena[8] allows inferred triples to be forward chained and data to be persisted in a relational database. However, Jena defines the schema for persistence so it is restricted to a triple store schema. Furthermore, Jena will require inference to be reprocessed in the event of updates to the dataset.

BitMat[11] is similar to our solution in that it uses compressed bit vectors and performs join operations using bit operations. BitMat combines these vectors into a complete in-memory bit matrix. BitMat cannot support additions and deletions without recreating the entire bit matrix. Their architecture requires upfront knowledge of the set of common subject and object URIs in order to generate the correct matrix indices necessary to perform subject-object joins. Therefore, additions cannot be supported because if a URI is added as a subject and object, the bit matrix would have to be recreated. Additionally, BitMat provides persistence by storing an image of its memory. BitMat is able to load this image fairly efficiently, but load time is still large enough that RDFKB will always achieve better performance time on cold runs.

For uncertainty reasoning, we have relied on the final report from [9], studying all of the different solutions and use cases described in this collaborative effort.

6 Future Work

The first area for future work is uncertainty reasoning. Existing research has proposed ontology definitions that include the probability that the inference applies.

We will add thresholds that allow vectors to be materialized with probability (1 in the vector will mean the triple is known with probability>=threshold). We plan to calculated all inferred triples, and probabilities at add time and stored, and support queries that select or rank by probability.

A second area for future work is cloud computing. In RDFVector, a significant portion of query time is spent accessing bit vectors. Once all the bit vectors have been retrieved, the remaining steps of the query plan, such as joins, can be performed in main memory through simple bit operations. Hadoop[10] could be used to distribute the bit vectors reads across a grid of computers, as each vector can be read separately. Join operations would be performed during the MapReduce processing.

7 Conclusion

We have proposed a solution for querying RDF datasets that is efficient and scalable. Our bit vector solution is novel and contributes significantly to the field. RDFVector stores inferred triples without a performance penalty. This enables simple inference queries with exceptional performance. Our experimental results show that our solution consistently outperforms vertical partitioning, triple store and RDF-3X. We tested 23 different queries on two large dataset benchmarks. The degree of performance improvement increases with the size of the dataset, showing that our solution is scalable. Inference queries provide even greater performance improvement, showing the viability of our inference solution.

Migrating to a cloud computing environment will further improve our scalability. Adding uncertainty reasoning will enable RDFVector to retrieve more information than available from state of the art RDF storage solutions.

8 References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In VLDB(2007) 411-422
2. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB(2008) 1008-1019
3. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. PVLDB(2008) 1553-1563
4. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. PVLDB(2008) 647-659
5. Lehigh University Benchmark (LUBM), http://swat.cse.lehigh.edu/projects/lubm
6. The Barton dataset, http://simile.mit.edu/wiki/Dataset:_Barton
7. MonetDB, http://monetdb.cwi.nl
8. Jena – A Semantic Web Framework for Java, http://jena.sourceforge.net
9. W3C Uncertainty Reasoning Incubator Group , http://www.w3.org/2005/Incubator/urw3
10. Hadoop, http://hadoop.apache.org
11. Atre, M., Hendler, J.A.: BitMat: A Main-memory Bit-matrix of RDF Triples. In ISWC(2009).
12. Costa, P.C.G.D., Laskey, K.B., Laskey, K.J.: PR-OWL: A Bayesian Ontology Language for the Semantic Web. In URSW (LNCS Vol.)(2008) 88-107