Materializing and Persisting Inferred and Uncertain Knowledge in RDF Datasets

James P. McGlothlin, Latifur Khan

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

Abstract

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. Advanced semantic web ontologies, especially in medicine and science, are developing. As more complex ontologies are developed, there is a 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. OWL (Web Ontology Language) defines rules that govern provable inference in semantic web datasets. In this paper, we detail a database schema using bit vectors that is designed specifically for RDF datasets. We introduce a framework for materializing and storing inferred triples. Our bit vector schema enables storage of inferred knowledge without a query performance penalty. Inference queries are simplified and performance is improved. Our evaluation results demonstrate that our inference solution is more scalable and efficient than the current state-of-the-art. There are also standards being developed for representing probabilistic reasoning within OWL ontologies. We specify a framework for materializing uncertain information and probabilities using these ontologies. We define a multiple vector schema for representing probabilities and classifying uncertain knowledge using thresholds. This solution increases the breadth of information that can be efficiently retrieved.

Introduction

The World Wide Web Consortium (W3C) defines the RDF data format as the standard mechanism for describing and sharing data across the web. All RDF datasets can be viewed as a collection of triples, where each triple consists of a subject, a property and an object. OWL inference rules allow queries and users to deduce additional knowledge from known facts. The goal of our research is to improve the efficiency and scalability of this information retrieval. The core strategy of our research is to apply all known inference rules to the dataset to determine all possible knowledge, and then store all this information in a relational database. Inference is performed at the time the data is added rather than at query time. All possible knowledge is materialized and persisted.

The simplest way to store RDF data in a relational database is to simpy create a three column table of RDF triples. However, research shows this is not the most efficient solution. Many alternative solutions have been proposed including vertical partitioning (Abadi et al., 2007), sextuple indexing (Weiss, Karras & Bernstein, 2008), and RDF-3X (Neumann&Weikum, 2008). None of these solutions store inferred data or address the task of querying inferred knowledge. Inference queries against these databases require detailed knowledge and encoding of the ontology logic, and require unions and joins to consolidate the inferred triples. There are existing solutions that perform inference in memory, providing simpler queries. There are even solutions which support inference and relational database storage, but they have fixed schema and do not support customized tables for efficiency. Such solutions pay a large query performance penalty due to increasing the dataset to include the inferred triples. We support inference at storage time combined with efficient database schema in a manner that not only simplifies queries but also improves performance and scalability. Our evaluation results show that our solution consistently outperforms the current state-of-the-art solutions for RDF storage and querying.

Our design relies on adding inferred RDF triples to the dataset. This would not be a viable solution unless these triples can be added and stored without increasing the performance cost of queries. We have designed a bit vector schema that can store these triples with negligible impact to query performance. Our bit vectors enable joins and unions to be performed as bitwise operations, and our inference materialization reduces the need for subqueries, joins, and unions at query time. The end result is that query performance is improved.

There are many motivations for storing uncertain information in semantic web repositories. The W3C Uncertainty Reasoning Incubator Group (URW3-XG) final report (2008, http://www.w3.org/2005/Incubator/urw3) defines 14 motivating use cases. There are also several proposals for representing probabilistic reasoning within semantic web ontologies. However, there is not an existing solution for storing the data and providing efficient queries. We present a framework for storing uncertain information and probability data. We materialize all inferred triples at addition time, even those involving probability. Every triple is persisted including its probability and an explanation of how it was inferred. We then define a multiple bit vector schema involving thresholds that will allow us to efficiently query this information and probabilities. Our contribution is that once the probabilities are calculated they can be persisted and made available for efficient querying.

Schema

In this section, we define our schema and tables. Figure 1 shows the process flow between tables. In this diagram, tables are represented by boxes with double lines. Actions that can change the data are represented by solid arrows. Actions that only query data are represented by dashed arrows.

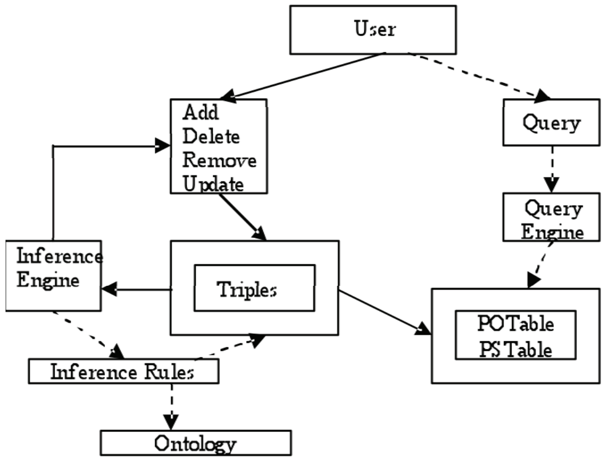

Figure 1: Process flow between tables

As this diagram shows, all additions and deletions are performed against the triples table. Therefore, this table is our *data management schema*. All queries are performed against our bit vector tables (POTable and PSTable), which provide our *query schema*. These tables and the process flow are described in detail in the following subsections.

Triples Table

We separate the schema exposed to the user from the schema used for queries. The most natural and common schema for RDF datasets is a triples table with columns for subject, property and object. Therefore, we provide such as table. This is the only table exposed to the inference rules and all user additions, deletions and updates are performed against this table. The triples table allows us to encapsulate our schema and to optimize processes for pushing updates across our bit vector tables.

The triples table also allows us to associate additional data with a triple. We have additional columns for probability, inference count, and inference origin. These columns are hidden from the user and managed by the inference engine. We use these columns to support deletion of inferred triples, to support explanation annotations, and to calculate probabilities.

One of the drawbacks of materialization is that materialized information is redundant and thus extra maintenance is required. Therefore, materialized information is generally read-only. We avoid this by making our bit vector tables read-only to the user but not to the system. We support both additions and deletions to the dataset via the triples table. These updates trigger automatic updates to the bit vector tables.

The triples table is not an efficient schema for queries. Moreover, a triples table schema suffers a query performance reduction whenever the number of triples is increased. Therefore, this schema would not allow us to store inferred triples efficiently. Therefore, the triples table is never used for queries. We supply a query interface to the user that translates the queries into query plans against our more efficient bit vector tables.

Bit Vector Tables

We store the RDF triples in two bit vector tables: the POTable and the PSTable. Each URI or literal is dictionary-encoded to a unique ID number. These ID numbers then are used as indices into our bit vectors.

The POTable includes five columns: the PropertyID, the SubjectID, the SubjectBitVector, the BitCount, and the Threshold. The SubjectBitVector has a 1 for every subject that appears with that property and object in a RDF triple. For example, all subjects matching <*?s type text*> can be retrieved from a single tuple and returned as a single bit vector, indexed by the ID numbers. The BitCount is the number of on bits in this vector. This column is used to support aggregation queries and to provide statistics used in our query optimization. The Threshold is used for querying uncertain knowledge associated with probabilities. Our threshold solution is explained in detail later in this paper. The PSTable contains four columns: the PropertyID, the SubjectID, the ObjectBitVector and the BitCount.

One of the key advantages of our bit vector schema is the capacity to use bitwise operations to perform joins. An RDF dataset can be viewed as one large three column triples table. In fact, this is the most natural schema, as we have already illustrated. Thus joins between triples are recursive joins against the same table, and there is little selectivity factor information available for query planning. Many semantic web queries involve many such joins, so they are often the bottleneck in query performance. Our BitCount information provides the missing selectivity factor information. More importantly, we can now perform many joins as efficient *and* and *or* operations. For example, consider the query "List all professors at University0". This can be performed by retrieving two bit vectors from POTable - one for property=type, object=Professor and one for property=worksFor, object=University0 - and then performing the *and* operation between the vectors.

The bit vectors are also the reason we are able to store inferred triples at little or no cost. The length of the bit vectors is equal to the max(ID). Each and every unique subject or object URI or literal in the dataset has a corresponding bit in these bit vectors.

Our inference solution relies upon adding additional triples for all inferred knowledge. Typical RDF databases incur a performance penalty for increasing the number of triples in the data set. However, except for a potential reduction in the achievable compression rate, our solution does not incur this penalty. Instead, we incur a performance penalty only when the dataset's vocabulary is increased. Our queries experience a performance reduction in a linear fashion corresponding to the number of unique URIs and literals in the dataset. This is because there is already a bit in the tables for each and every possible combination of the known subjects, properties and objects. Many times, inferred triples will not introduce unique URIs at all, and if unique URIs are introduced, they are only unique for that ontology rule. While an RDF dataset may include millions of triples, the number of unique terms in the ontology is generally not large. For our experiments, we utilize the LUBM (Lehigh University Benchmark, http://swat.cse.lehigh.edu/projects/lubm/) dataset with more than 44 million RDF triples. For this dataset, 20,407,385 additional triples are inferred, yet only 22 unique URIs are added by this inference. Thus, there is no performance penalty for the addition of the millions of inferred triples, only for the addition of 22 new URI terms. For these reasons, our bit vector solution allows inferred triples to be added to the dataset at almost no performance cost.

The bit vectors can get very large. For the Barton dataset (http://simile.mit.edu/wiki/Dataset:_Barton), there are 18,468,653 unique URIs and literals. However, these vectors tend to be very sparse. The largest BitCount for the Barton dataset is for *type text* which has 1,542,480 matches. Even in this case 0s out number 1s in the vector 12 to 1, allowing for exception compression. We use the D-Gap compression scheme (http://bmagic.sourceforge.net/dGap.html). The key advantage to D-Gap is that bitwise operations can be performed against the vectors without decompression. Our results show that compressing the vectors minimizes their impact on memory and storage consumption.

Inference

Our design is to add triples to the dataset at addition time. Therefore, we must have a process for doing this. One of our design goals is to allow each inference rule to be implemented and registered separately. Therefore, we are not tied to a particular ontology representation. Another of our design goals is to not require the inference rule to have knowledge of our schema. Therefore, our inference rules execute against the triples table only. Another design goal is that inference rules only have to handle one level of inference. Therefore, we have designed a recursive solution. Another design goal is that inferred triples are properly deleted when the base triple is deleted. We specify our solution for deletions in this section as well.

Figure 1 shows the process flow for inference. When a triple is added the inference engine is called. It queries the inference rules in order to determine, via forward-chaining, any additional triples that can be inferred. These inferred triples are then added to the triples table, by the same add method as used by the user. This cycle demonstrates the recursive nature of our solution. Deletion is actually no different from the inference rule's perspective. The inference rule determines the triples than can be forwarded-chained, and then the inference engine deletes, rather than adds, the inferred triples.

Let us consider an example of how our solution works. Consider the triple *<Professor0 type FullProfessor>*. The LUBM ontology will allow us to infer 4 additional triples: *<Professor0 type Professor>, <Professor0 type Faculty>, <Professor0 type Employee>, <Professor0 type Person>*. Our strategy is to materialize and store all 4 inferred triples. As *Professor, Faculty, Employee* and *Person* exist elsewhere in the dataset, no new vocabulary is introduced. Therefore, all that is required to add these triples is to change 0s to 1s in the bit vector. The size of the database and the bit vectors is not increased. Now, we can execute a query such as *List all persons* by reading a single bit vector. *Person* has 21 subclasses in the LUBM ontology. Therefore, to query all persons with vertical partitioning or RDF-3X would require 21 subqueries and 20 unions.

Now, we will address the specifics for how the inferred triples are added and managed. We provide an inference engine. Each inference rule registers itself using Java factory methods. Each inference rule provides a method *forwardChain(subject,property,object)*. When a triple is added, the inference engine iterates through the inference rules and gives each inference rule the opportunity to forward-chain additional inferred triples based on the base triple being added. The only change an inference rule can make to the dataset is to add more triples. This is a recursive solution. In the above example, the subClassOf inference rule will add *<Professor0 type Professor>* when *<Professor0 type FullProfessor>* is added. Then the subClassOf inference rule is executed again for the new addition, and adds *<Professor0 type Faculty>*.

We have implemented inference rules to support the OWL constructs subClassOf, sameAs equivalentClass, subPropertyOf, TransitiveProperty, SymmetricProperty, inverseOf, and intersectionOf. Additionally, we implemented an inference rule to support the *Records* inference relationship defined by the Barton Dataset.

Deletions offer a special challenge for inferred triples. If the base triple is deleted and the inferred triple no longer applies, it should be deleted as well. However, a triple might be inferred from more than one base triple. For example, assume John is a graduate student and a research assistant in computer science at MIT. Now, John gets an internship at IBM, and they agree to pay for the rest of his education. He is no longer a research assistant or an employee and no longer works for the department. However, he is still a student and a member of the school, because these facts can still be inferred from the fact John is a graduate student at MIT. This problem is solved by maintaining an inference count. John is a memberOf MIT because he works for the CS department and because he is a graduate student at MIT. If John stops working for the CS department, he will still be a member of MIT.

Our solution is to maintain an InferenceCount column in the triple table. When a triple is added by inference, the InferenceCount is incremented. When a triple is deleted, we rerun the inference rules as though the triple was being added. However, instead of adding inferred triples, we decrement the InferenceCount for the inferred triple. If the InferenceCount becomes 0, we delete the inferred triple.

There is also research in coherency theory and belief revision (Flouris et al., 2008) that suggests that inferred triples should survive the removal of the base triples. An example is medical data. We would like to gather statistics and make inferences across patient records. However, for privacy reasons, the patient's records must be deleted from the dataset. In such instances, inferred triples should not be removed when concrete triples are removed. We solve this problem by providing two separate user functions. *Delete* deletes a triple and all relevant inferred triples, *remove* deletes only the concrete triple specified.

Uncertainty Reasoning

OWL has provable inference. All inferred triples are known to be true with absolute certainty. Uncertainty reasoning addresses the issue of knowledge that might be true. Our goal is to associate probability numbers with facts (RDF triples) and to enable queries based on probability conditions. In order to make these queries efficient, our goal is to materialize and store the inferred facts and associated probabilities.

Uncertainty is too broad a subject to claim an all encompassing solution. The URW3-XG final report defines two kinds of uncertainty. We wish to concentrate on the first, "uncertainty inherent to the data". URW3-XG details 14 use cases. We looked most carefully at use case 5.4 "Ontology Based Reasoning and Retrieval from Large-Scale Databases" and use case 5.14 "Healthcare and Life Sciences". The key benefits of our solution are scalability and query efficiency. Therefore, we concentrated on uncertainty that appears within the data model for large scale databases. We have attempted to make our design as extensible as possible, but we are not trying to create our own probabilistic reasoning system. Our contribution is that we allow probabilities to be stored and retrieved with the RDF triples efficiently. We assert that this will enable scalable and efficient probabilistic reasoning systems.

Ontology Representation

Our OWL inference rules work under the simple logical concept (A B) & A ∴B. The inference rule from the ontology represents A B, the base triple is A and the inferred triple is B. The only difference for our uncertainty reasoning solution is that we can associate a probability.

There is no established standard for representing probability in a semantic web ontology. URW3-XG defines 5 approaches to uncertainty in the web: description logic, Bayesian networks, first order logic, fuzzy logic and belief networks. Our goal is not to restrict the ontology representation. Therefore, as with provable inference, we allow inference rules to register and we call them during the addition of triples. They check for conditionals, determine triples to add, calculate the probability for the inferred triples, and add this information. As with provable inference, all inferred triples are materialized and stored at addition time.

Schema

Triples Table. Our design is to support probability within our vectors. However, in order to not penalize queries without probability, and to not absorb an unlimited amount of memory and storage, we need to limit the precision of the probabilities identified via the vectors. Therefore, our design is to also include the exact probability in the triples table as an extra column. Furthermore, we include an explanation of the inference in the triples table. In support of uncertainty reasoning, we add explanations to each inferred triple, whether or not it has probability. An example of a triples table with this annotation column is shown in Table 1.

Table 1: Example triples table with explanation column

Subject	Property	Object	Prob	Explanation
Professor0	type	FullProfessor	1	
Professor0	type	Professor	1	*subclassOf(Professor FullProfessor, Professor0 type FullProfessor)*
Professor0	type	Faculty	1	*subclassOf(Professor Faculty, Professor0 type Professor)*

Vectors. We examined two options for storing probabilities within our vectors. The first was to replace the bit vectors with vectors of 2 bits, 4 bits or 16 bits depending on the required level of precision. Known facts would be stored as all 1s and known non-facts as all 0s. Uncertain facts would be represented as fractions between 1 and 0. For example, with 2 bits, we could represent probability=1 with (11), probability=0 with (00), 0.5 >=probability >0 with (01) and 1>probability>0.5 with (10). We rejected this solution for a few reasons. This solution would increase the size of every vector whether or not it included uncertain information. Furthermore, it would jeopardize the ability to perform bitwise operations to execute joins and unions. Finally, it would not make it easy to perform selections based on probability.

Instead, we adopt a multiple vector system with threshold. This solution adds a column to the POTable, the threshold column. The vectors are bit vectors with 1s for every triple known with probability>=threshold. For vectors without probability, the threshold is 1. Non-probabilistic queries always use vectors with threshold=1 and they are unaffected. The trade-off is extra storage space because we will store multiple vectors with multiple thresholds. However, memory consumption is reduced because we only need to read in the vector with the appropriate threshold.

Thresholds. Now that we have chosen the multiple vector approach, we examine thresholds. Suppose we want a list of subjects that are likely to have lung cancer. Inference rules could be based on prior conditions, tobacco use, family history, age, employment, geography, last doctor visit, etc. For every such inference rule known, we will have executed the inference and materialized the probability. An example POTable for this scenario is shown in Table 2.

Table 2: Example POTable with thresholds

Property	Object	Threshold	SubjectBitVector	Bit Count
hasDisease	LungCancer	1.0	1010010101001000100	7
hasDisease	LungCancer	0.9	1011010101011000100	9
hasDisease	LungCancer	0.8	1011010101011000101	10
hasDisease	LungCancer	0.7	1111010101011010111	13
hasDisease	LungCancer	0.6	1111010101011010111	13
hasDisease	LungCancer	0.5	1111010101011010111	13
hasDisease	LungCancer	0.4	1111010101011110111	14
hasDisease	LungCancer	0.3	1111010101011110111	14
hasDisease	LungCancer	0.2	1111011101011110111	15
hasDisease	LungCancer	0.1	1111011101011110111	15

We can query all of those with lung cancer (probability=1.0) the same as we could before probability. We can now also retrieve all those with >0.4 chance of having lung cancer by reading one tuple. We can even retrieve all those with >0.4 and <0.5 probability, by reading two tuples and executing the bitwise operation *and not* between them. These vectors can now be joined with other vectors (example: Age over 40) using bitwise operations against the vectors.

Also, notice in our example that the bit count allow us to determine how many subjects match the probability conditional without even reading the vector. Also, notice in our example, the were no changes between some of the thresholds. The tuples with thresholds 0.6, 0.5, 0.3 and 0.1 can be deleted since they offer no new information; we can use the higher threshold and get the same results. This allows us to save storage.

If a query requests a different probability, such as 0.45, that is not represented by a threshold, we provide a two part solution. First, note this problem does not necessarily have to be solved. Many ontology representations restrict the list of probabilities values to defined sets such as low, medium, high, etc. Therefore, the thresholds are known and every threshold can be provided a bit vector. However, we nonetheless solve this problem for the generic case. A query which wanted all those with probability of lung cancer > 0.45 would first query the 0.5 threshold. These subjects definitely belong to the result. Then it would retrieve the list of subjects with probability >0.4 and <0.5 using the *and not* operation as specified above. Finally, for each such triple it would query the triples table to get the exact probability. If the probability was greater than 0.45 it would add the subject to the results. Thus, the vectors are used for efficient selection, and the triples table is available for precise comparisons.

There are many options for setting thresholds. In our example, we set the thresholds at 0.1 intervals. However, our solution is to allow a user option that specifies the threshold interval. The key advantage of this is that it allows the user to know the thresholds and thereby specify the most efficient queries.

Probability Propagation

We propagate probabilities throughout the system based on the simple Bayesian logic rule that *P(A)= P(A|B)*P(B)*. Therefore, inference rules for provable OWL inference (thus *P(A|B)=1.0)* add the inferred triple with the same probability as the base triple. Probability inference rules use the same formula, thus factoring the probabilities together when the base triple is also uncertain.

Ambiguities. Bayesian networks are a complex area of study. Some facts are mutually exclusive. For example a person is either male or female. Therefore if P(A|Male)=0.5 and P(A|Female)=0.1 we can conclude P(A|Person) =0.5*0.5 + 0.5*0.1= 0.30. However, if the probability of Person1 having stomach cancer is 0.1 and the probability of Person1 having lung cancer is 0.1, we cannot conclude the probability of Person1 having cancer is 0.2, because the person could have both. We would need the probability of the person have lung and stomach cancer. Furthermore, we need to handle non-monotonistic reasoning, such as the classic penguin example. Even though birds have a 0.95 probability of flight ability and a penguin is a bird, the penguin overrides this probability by specifying penguins have a 0.0 probability of flight ability.

We wish an architectural framework that allows decisive solutions to such ambiguities. Therefore we support a conflict handler. In addition to the *forwardChain* method, the inference rules can register a *resolveProbability* method. When an inferred triple already exists, and an inference rule attempts to add the same triple with a new probability, *resolveProbability*. is executed to determine the final probability. The explanation column of the triples table helps us implement conflict handlers. For example, a conflict handler can choose the probability that has the shortest inference path to a concrete triple. This would support override behavior such as the penguin example.

Here is a generic example of probability resolution. Consider an ontology that defines two rules P(A|B) and P(A|C). Now, the user adds triple B to the dataset. The inference engine will forward-chain triple A and add it to the dataset with a probability, P(A|B). Now, the user adds triple C to the dataset. The inference engine will forward-chain A and attempt to add it with P(A|C). However, A already exists with P(A|B). Therefore, *resolveProbability* will be called to determine the probability of triple A. For example, an inference rule can be implemented using the Bayesian reasoning technology from BayesOWL (Zhang et al., 2009). This inference rule can then use the Bayesian network to determine P(A|B^C) which in fact is the final probability of triple A.

Our goal in this framework is not to solve such ambiguities anymore than it is to define ontology representations. Significant research already exists in these areas, and our design is to allow these solutions to be implemented within our framework. The contribution of our work is that once the probabilities are calculated they can be persisted and made available for efficient querying.

Evaluation Results

We identified two large dataset benchmarks used in other evaluation research: LUBM and the Barton Dataset. LUBM is a synthetic dataset for a university domain. We created a database using the LUBM dataset with 400 universities and 44,172,502 tuples. The Barton Dataset is an RDF representation of MIT's library catalog. There are over 50 million unique triples in the Barton Dataset.

LUBM has a well-defined OWL ontology which expresses a large number of inference rules. LUBM defines 14 test queries, and 12 of those 14 queries involve inference. Barton Dataset has 12 queries as specified in the evaluation paper by Sidirourgos at al. (2008). Barton Dataset only defines one inference rule, and only 3 of the queries involve inference. For our analysis, we will specify summary information about all the queries, and specifics only about some of the queries involving inference.

All of our experiments here were performed on a system with an Intel Core 2 Duo CPU @ 2.80 GHz, with 8GB RAM. We used the MonetDB (http://monetdb.cwi.nl/) column store database for our persistence. We ran both hot and cold runs, and used the 64-bit Ubuntu 9.10 operating system.

In prior research, the approach has been to either avoid the queries that include inference or to support these queries by translating them into unions of subqueries without inference. We support these queries as defined, without requiring the query to have any knowledge of the inference. There is no high-performance solution with inference that uses relational databases for RDF datasets. It seems uninformative to compare our results with solutions that access and parse RDF documents directly, and the performance of such tools is slower by many orders of magnitude. Therefore, we compare with vertical partitioning, RDF-3X and triple store solutions. None of these solutions provides automated support for inference. Instead, the query implementation requires specific knowledge and encoding of the ontology logic. These queries require a minimum of 4 subqueries and 3 unions, and a maximum of 29 subqueries. However, we also tested adding the inferred triples to the datasets in these solutions. While this did simplify the queries, it actually reduced the query performance. This is because these schemata pay a performance penalty for increasing the size of the dataset.

Our solution outperformed RDF-3X, vertical partitioning and triple store solutions for all 14 LUBM queries. For Barton Dataset, our solution achieved the best performance for all 12 queries on cold runs, and for 11 out of 12 queries on hot runs. This includes all 3 inference queries, and we achieved the greatest performance improvement for these queries. This demonstrates the viability of our inference materialization solution.

The complete query time across all 14 LUBM queries for our solution was 8.1 seconds. Vertical partitioning required 68.8 seconds, RDF-3X 22.7 seconds, and triple store (with SPO ordering) 113.9 seconds. Summary statistics for Barton Dataset are included in the Table 3. PSO indicates a triple store solution, sorted by *property, subject, object,* and implemented with MonetDB. SPO indicates the same solution sorted by *subject, property and* object. We achieved the best average query time for both hot and cold runs. In this table. our solution is labeled MBV for "materialized bit vectors" .

Table 3: Average query times (in seconds) for Barton dataset

	MBV	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

We will now provide details for one LUBM query and one Barton query.

LUBM Query 5

List persons who are members of a particular department

This query invokes five kinds of inference rules: subClassOf, subPropertyof, inverseOf, TransitiveProperty and intersectionOf. There are 21 classes in the Person hierarchy in the LUBM ontology. In the LUBM dataset, there are instantiations for 8 of these classes. Therefore, to query *?x type Person* using vertical partitioning or RDF-3X requires 21 subqueries; 8 of which will return results that must be unioned together. There are three different ways to express memberOf in the dataset. It can be expressed directly with the memberOf property, or through the subProperty worksFor, or through the inverse property member. The range of the memberOf property is type Organization which is affected by the transitive property subOrganizationOf. So it is necessary to query if the subject is a memberOf or worksFor any entity that in turn is a subOrganizationOf the department. Transitive closure is not limited to a single level, therefore it is necessary to repeat this loop until the organization does not appear as a suborganization.

Our solution materialized the inferred triples at addition time so our query does not need to address this inference logic. We implement this query by simply retrieving the bit vector from the POTable for *property=type, object=Person*, retrieving the bit vector from the POTable for *property=memberOf*, *object=Department0,* and then executing the bit operation *and* between these two bit vectors. We are able to perform this query for the 44 million triples dataset in 0.23 seconds. The next best solution, RDF-3X, requires 2.88 seconds and vertical partitioning requires 6.28 seconds.

Barton Dataset Query 5

This query defines a simple inference rule: *(X Records Y) & (Y Type Z)* *(X Type Z)*. The query lists the subject and inferred type of all subjects with origin DLC.

We implemented and registered this inference rule. Thus, the type of X is determined at add time and stored in the database. This simplifies the query, eliminating a subquery. Our query engine implements this query using the PSTable and the POTable. Our cold run time, 0.41 seconds, was faster than all other solutions by a factor of 4. Our hot run time, 0.21 seconds, was faster by a factor of 2. This clear performance improvement shows the viability of our inference solution.

Uncertainty Reasoning Results

For uncertainty reasoning, we have not yet been able to identify a benchmark dataset and ontology to evaluate our solution for scalability and efficiency. We have tested with ontology representations using BayesOWL and description logic (Pronto, http://pellet.owldl.com/pronto). We have used example datasets and small ontologies to validate our solution for correctness.

We have also created our own simple experiments. We added an arbitrary property to LUBM with probability. We defined it as property="”has” and object=”possible”. We did not create an inference rule, we simply randomly assigned 1,000,000 subjects of type Person *has possible* with probabilities randomly assigned between 0 and 1. Then, we changed LUBM Query 1 to add “and *has possible* probability >=0.4.” The query times for LUBM Query 1 were all less than 2ms slower than the query time without probability. As our LUBM query time was more than 2ms faster than all other solutions, this allows us to conclude that our probability solution is also faster.

In addition, we created our own test dataset. We defined inference rules that determine the probability that a person will develop skin cancer. We defined rules based on family history, number of moles, hair color, age, sex, exposure to sun, previous burns, and previous cancer history, using statistics obtained from the Internet. We then created arbitrary persons (just named Person1, etc) and arbitrarily assigned them values for some or all of these fields. We created 5 million such persons, with a total of 20 million triples. We were able to query the list of people with the likelihood of getting skin cancer over 70% in 48ms. We were able to provide summary statistics of the number of persons in each range (for example number of people with >=50% likelihood to get skin cancer) in 4 ms. We were able to rank all the persons by their likelihood to get skin cancer in 562 ms. These results demonstrate that our solution is able to query probability without any degradation in performance.

Trade-Offs

We have identified four trade-offs incurred by our solution: the time to initially populate the database, the time to perform updates to the dataset, the database's physical size, and memory consumption. To verify the extent of these trade-offs, we tested and quantified all four.

The time to initially populate the database with the Barton dataset is 41 minutes. The time required to add triples to the database depends on the nature of the triples added. We tested the time to add 1 million triples to the LUBM dataset, adding whole universities with complete sets of data. This took 3.4 minutes compared to 1.7 minutes for the fastest solution, vertical partitioning. To show how the data affects the times, we added 1 million students to the database. This addition took only 1.1 minutes because fewer bit vectors were affected. Additionally, we did a smaller test, the addition of just 10,000 triples. This addition required only 2.9 seconds.

There is no established method for testing the performance times of deletions. We were able to delete a million triples in as little as 8 seconds depending on the data chosen. To get more accurate results, we used a method similar to cross validation. We chose a 40 million triples dataset, and then we chose 1 million of the triples at random using the ID numbers. We performed this test 10 separate times with 10 different sets of deletions. The average time to delete 1 million triples was 26.7 seconds.

To performed all the tests in this paper, our maximum memory consumption was 3.54 GB of RAM. The size of the database was 3.4 GB for the Barton dataset. Note that one trade-off we chose in order to limit storage space is that we only store uncertain information in the POTable. This limitation does not reduce functionality, because we can always use the *inverseOf* construct to convert the data definitions. For example, *s member o* can be expressed as *o memberOf s.* So, this limitation defines how we define the ontology, but it does not limit the functionality.

We assert that nominal trade-offs in memory usage, storage size, and initial population time are not prohibitive. By storing data in multiple formats and materializing inferred triples, we increase storage requirements but we reduce I/O. Since we support additions and deletions, the initial populating of the dataset is a one-time event. The increase in physical storage space is not significant, and hard drive space is affordable in today's market.

Related Work

Significant research has addressed efficient schemata and storage schemes for RDF data. However, our solution is novel in that we use bit vectors, and we persist inferred triples with high performance.

Vertical partitioning divides the dataset into 2 column tables (subject, object) based on property. This enables high speed subject-subject joins. Hexastore uses six indices to manage RDF data. RDF-3X uses aggregate indexes, join statistics and query optimization to support efficient queries against RDF datasets. BitMat (Atre&Hendler, 2009) uses bit vectors as well but combines them into a main memory bit matrix. However. BitMat cannot support additions or updates.

Jena supports registering and executing inference rules. However, the inferred triples are not persisted, and it isn't possible to update the dataset without re-executing inference from scratch. Lu et al. (2005) propose materializing and storing inferred triples. However, their schema incurs a performance penalty for storing these triples, and the query performance of this solution is not competitive.

There is no standard for representing uncertainty in OWL ontologies. We have relied heavily on the URW3-XG's reports, recommendations, and use cases. We have examined popular representation formats including Pronto, PR-OWL(Costa, Laskey & Laskey, 2008), and BayesOWL. We have utilized available tools including Pronto and UnBBayes (http://unbbayes.sourceforge.net). We are examining fuzzy logic solutions including DeLorean (Bobillo&Delgado, 2008) and FiRE (http://www.image.ece.ntua.gr/~nsimou/FiRE/).

Conclusion and Future Work

We have defined a bit vector schema that is efficient and scalable. We have detailed a framework for materializing inferred triples at addition time and storing them using this schema. We have shown that this simplifies inference queries and drastically improves performance. We have presented query performance results using accepted benchmark datasets that demonstrate our solution outperforms the current state-of-the-art. We have presented a solution which adds support for materialization and persistence of uncertain data and probabilities. We have validated the correctness and efficiency of this solution.

There is no established data representation for uncertainty in semantic web databanks. Therefore, there are no large benchmark datasets or well defined queries to test and evaluate scalability and efficiency of queries involving probabilistic reasoning. We have evaluated our solution with simple manufactured test datasets. In the future, we will evaluate and optimize our solution for efficiency and scalability. We hope to identify or create benchmarks that correlate to the use cases defined in the URW3-XG report. We will also try to improve on our extensibility, to test with more probabilistic reasoning systems, to further integrate with available tools, and to support uncertainty from other sources such as ontology mapping. Finally, we hope to integrate with Hadoop (http://hadoop.apache.org/) to make an even more scalable solution by using cloud computing.

References

Abadi, D.J.; Marcus, A.; Madden, S.; Hollenbach, K.J. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. *In Proc. of VLDB*, 411-422.

Weiss, C.; Karras, P.; Bernstein, A. 2008. Hexastore: sextuple indexing for semantic web data management. *In Proc. of VLDB*, 1008-1019.

Sidirourgos, L.; Goncalves, R.; Kersten, M.L.; Nes, N.; Manegold, S. 2008. Column-store support for RDF data management: not all swans are white. *In Proc. of VLDB*, 1553-1563.

Neumann, T.; Weikum, G. 2008. RDF-3X: a RISC-style engine for RDF. *In Proc. of VLDB*, 647-659.

Lu, J.; Yu, Y.; Tu, K.; Lin, C.; Zhang, L. 2005. An Approach to RDF(S) Query, Manipulation and Inference on Databases. *In Proc. of WAIM*, 172-183.

Costa, P.C.G.D.; Laskey, K.B.; Laskey, K.J. 2008. PR-OWL: A Bayesian Ontology Language for the Semantic Web. *In Proc. of URSW*, 88-107.

Yang, Y.; Calmet, J. 2005. OntoBayes: An Ontology-Driven Uncertainty Model. *In Proc. of CIMCA/IAWTIC*, 457-463.

Ding, Z.; Peng, Y. 2004. A Probabilistic Extension to Ontology Language OWL. *In Proc. of HICSS*.

Lukasiewicz, T. 2008. Expressive Probabilistic Description Logics. *Artificial Intelligence*, *172(6-7),* 852-883.

Bobillo, F.; Delgado, M.; Gomez-Romero,J. 2008. DeLorean: A Reasoner for Fuzzy OWL 1.1. *In Proc. of URSW*.

Cali, A.; Lukasiewicz, T.; Prediou, L.; Stuckenschmidt, H. 2007. A Framework for Representing Ontology Mappings under Probabilities and Inconsistency. *In Proc. of URSW*.

Flouris, G; Fundulaki, I; Pediaditic, P; Theoharis, Y; Christophides, V. 2009. Coloring RDF Triples to Capture Provenance. *In Proceedings of ISWC,* 196-212.

Atre, M; Hendler, J. 2009. BitMat: A Main-memory Bit-Matrix of RDF Triples. *In Proceedings of SSWS Workshop, ISWC.*

Zhang, S; Sun, Y; Peng, Y; Wang, X. 2009. BayesOWL: A Prototypes System for Uncertainty in Semantic Web. *In Proceedings of IC-AI,* 678-684.

Bobillo, F.; Delgado, M.; Gomez-Romero,J. 2008. DeLorean: A Reasoner for Fuzzy OWL 1.1. *In Proc. of URSW*.