# Query Optimization in Cooperation with an Ontological Reasoning Service

Hui Shi
School of Computer and Information
Hefei University of Technology
Hefei, China
hshi@cs.odu.edu

Kurt Maly,Steven Zeil
Department of Computer Science
Old Dominion University
Norfolk, USA
{maly,zeil}@cs.odu.edu

***Abstract*—Interposing a backward chaining reasoner between a knowledge base and a query manager yields an architecture that can support reasoning in the face of frequent changes. However, such an interposition of the reasoning introduces uncertainty regarding the size and effort measurements typically exploited during query optimization. This paper presents an algorithm for dynamic query optimization in such an architecture. Experimental results confirming its effectiveness are presented.**

***Keywords-semantic web; ontology; reasoning; query optimization; backward chaining***

## I. INTRODUCTION

Consider a potential chemistry Ph.D. student who is trying to find out what the emerging areas are that have good academic job prospects. What are the schools and who are the professors doing groundbreaking research in this area? What are the good funded research projects in this area? Consider a faculty member who might ask, "Is my record good enough to be tenured at my school? At another school?" It is possible for these people each to mine this information from the Web. However, it may take a considerable effort and time, and even then the information may not be complete, may be partially incorrect, and would reflect an individual perspective for qualitative judgments. Thus, the efforts of the individuals neither take advantage of nor contribute to others' efforts to reuse the data, the queries, and the methods used to find the data. We believe that qualitative descriptors such as "groundbreaking research in data mining" are likely to be accepted as meaningful if they represent a consensus of an appropriate subset of the community at large. Once accepted as meaningful, these descriptors can be realized in a system and made available for use by all members of that community.

The system implied by these queries is an example of a semantic web service where the underlying knowledgebase covers linked data about science research that are being harvested from the Web and are supplemented and edited by community members. The query examples given above also imply that the system not only supports querying of facts but also rules and reasoning as a mechanism for answering queries.

A key issue in such a semantic web service is the efficiency of reasoning in the face of large scale and frequent change. Here, scaling refers to the need to accommodate the substantial corpus of information about researchers, their projects and their publications, and change refers to the dynamic nature of the knowledgebase, which would be updated continuously.

In semantic webs, knowledge is formally represented by an ontology as a set of concepts within a domain, and the relationships between pairs of concepts. The ontology is used to model a domain, to instantiate entities, and to support reasoning about entities. Common methods for implementing reasoning over ontologies are based on First Order Logic, which allows one to define rules over the ontology. There are two basic inference methods commonly used in first order logic: forward chaining and backward chaining [1].

A question/answer system over a semantic web may experience changes frequently. These changes may be to the ontology, to the rule set or to the instances harvested from the web or other data sources. For the examples discussed in our opening paragraph, such changes could occur hundreds of times a day. Forward chaining is an example of data-driven reasoning, which starts with the known data in the knowledgebase and applies modus ponens in the forward direction, deriving and adding new consequences until no more inferences can be made. Backward chaining is an example of goal-driven reasoning, which starts with goals from the consequents, matching the goals to the antecedents to find the data that satisfies the consequents. As a general rule forward chaining is a good method for a static knowledgebase and backward chaining is good for the more dynamic cases.

The authors have been exploring the use of backward chaining as a reasoning mechanism supportive of frequent changes in large knowledge bases. Queries may be composed of mixtures of clauses answerable directly by access to the knowledge base or indirectly via reasoning applied to that base. The interposition of the reasoning introduces uncertainty regarding the size and effort associated with resolving individual clauses in a query. Such uncertainty poses a challenge in query optimization, which typically relies upon the accuracy of these estimates. In this paper, we describe an approach to dynamic optimization that is effective in the presence of such uncertainty.

In section II, we provide background material on the semantic web, reasoning, and database querying. Section 3 formally gives the overall algorithm for answering a query. The details of the optimization methods we have developed within the backward chaining algorithm will be described in a later paper. We have implemented this algorithm and performed experiments with data sets ranging from 1 million to 6 million facts. In section 4 we report on some of these experiments, comparing our new algorithm with a commonly used backward chaining algorithm JENA [2].

## II. Related Work

A number of projects (e.g., Libra [3, 4], Cimple [5], and Arnetminer [6]) have built systems to capture limited aspects of community knowledge and to respond to semantic queries. However, all of them lack the level of community collaboration support that is required to build a knowledge base system that can evolve over time, both in terms of the knowledge it represents as well as the semantics involved in responding to qualitative questions involving reasoning.

Many knowledge bases [7-10] organize information using ontologies. Ontologies can model real world situations, can incorporate semantics which can be used to detect conflicts and resolve inconsistencies, and can be used together with a reasoning engine to infer new relations or proof statements.

Two common methods of reasoning over the knowledge base using first order logic are forward chaining and backward chaining [1]. Forward chaining is an example of data-driven reasoning, which starts with the known data and applies modus ponens in the forward direction, deriving and adding new consequences until no more inferences can be made. Backward chaining is an example of goal-driven reasoning, which starts with goals from the consequents matching the goals to the antecedents to find the data that satisfies the consequents. Materialization and query-rewriting are inference strategies adopted by almost all of the state of the art ontology reasoning systems. Materialization means pre-computation and storage of inferred truths in a knowledge base, which is always executed during loading the data and combined with forward-chaining techniques. Query-rewriting means expanding the queries, which is always executed during answering the queries and combine with backward-chaining techniques.

Materialization and forward chaining are suitable for frequent computation of answers with data that are relatively static. Owlim [11] and Oracle 11g [12], for example implement materialization. Query-rewriting and backward chaining are suitable for efficient computation of answers with data that are dynamic and infrequent queries. Virtuoso [13], for example, implements a mixture of forward-chaining and backward-chaining. Jena [2] supports three ways of inferencing: forward-chaining, limited backward-chaining and a hybrid of these two methods.

In conventional database management systems, query optimization [14] is a function to examine multiple query plans and selecting one that optimizes the time to answer a query. Query optimization can be static or dynamic. In the Semantic Web, query optimization techniques for the common query language, SPARQL [15, 16], rely on a variety of techniques for estimating the cost of query components, including selectivity estimations [17], graph optimization [18], and cost models [19]. These techniques assume a fully materialized knowledge base.

Benchmarks evaluate and compare the performances of different reasoning systems. The Lehigh University Benchmark (LUBM) [20] is a widely used benchmark for evaluation of Semantic Web repositories with different reasoning capabilities and storage mechanisms. LUBM includes an ontology for university domain, scalable synthetic OWL data, and fourteen queries. The University Ontology Benchmark (UOBM) [21] extends the LUBM benchmark in terms of inference and scalability testing. Both LUBM and UOBM have been widely applied to the state of the art reasoning systems to show the performance regarding different aspects [20, 21].

## III. Dynamic Query Optimization with an Interposed Reasoner

A query is typically posed as the conjunction of a number of clauses. The order of application of these clauses is irrelevant to the logic of the query but can be critical to performance.

In a traditional data base, each clause may denote a distinct probe of the data base contents. Easily accessible information about the anticipated size and other characteristics of such probes can be used to facilitate query optimization.

The interposition of a reasoner between the query handler and the underlying knowledge base means that not all clauses will be resolved by direct access to the knowledge base. Some will be handed off to the reasoner, and the size and other characteristics of the responses to such clauses cannot be easily predicted in advance, partly because of the expense of applying the reasoner and partly because that expense depends upon the bindings derived from clauses already applied. If the reasoner is associated with an ontology, however, it may be possible to relieve this problem by exploiting knowledge about the data types introduced in the ontology..

In this section, we describe an algorithm for resolving such queries using dynamic optimization based, in part, upon summary information associated with the ontology. In this algorithm, we exploit two key ideas: 1) a greedy ordering of the proofs of the individual clauses according to estimated sizes anticipated for the proof results, and 2) deferring joins of results from individual clauses where such joins are likely to result in excessive combinatorial growth of the intermediate solution.

We begin with the definitions of the fundamental data types that we will be manipulating. Then we discuss the algorithm for answering a query. A running example is provided to make the process more understandable.

We model the knowledge base as a collection of triples. A triple is a 3-tuple (x,p,y) where x, p, and y are URIs or constants and where p is generally interpreted as the identifier of a property or predicate relating x and y. For example, a knowledge base might contains triples

```
(Jones, majorsIn, CS), (Smith, majorsIn, CS),
(Doe, majorsIn, Math),  (Jones, registeredIn, Calculus1),
(Doe, registeredIn, Calculus1).
```

A QueryPattern is a triple in which any of the three components can be occupied by references to one of a pool of entities considered to be variables. In our examples, we will denote variables with a leading '?'. For example, a query pattern denoting the idea "Which students are registered in Calculus1?" could be shown as

(?Student,registeredIn,Calculus1).

A query is a request for information about the contents of the knowledge base. The input to a query is modeled as a sequence of QueryPatterns. For example, a query "What are the majors of students registered in Calculus1?" could be represented as the sequence of two query patterns

[(?Student,registeredIn,Calculus1),
(?Student, majorsIn, ?Major)].

The output from a query will be a QueryResponse. A QueryResponse is a set of functions mapping variables to values in which all elements (functions) in the set share a common domain (i.e., map the same variables onto values). Mappings from the same variables to values can be also referred to as variable bindings. For example, the QueryResponse of query pattern (?Student, majorsIn, ?Major) could be the set

{{?Student => Jones, ?Major=>CS},
{?Student => Smith, ?Major=>CS },
{?Student => Doe, ?Major=> Math }}.

The SolutionSpace is an intermediate state of the solution during query processing, consisting of a sequence of (preliminary) QueryResponses, each describing a unique domain. For example, the SolutionSpace of the query "What are the majors of students registered in Calculus1?" that could be represented as the sequence of two query patterns as described above could first contain two QueryResponses:

[{{?Student => Jones, ?Major=>CS},
{?Student => Smith, ?Major=>CS },
{?Student => Doe, ?Major=> Math }},
{{?Student => Jones},{?Student => Doe }}]

Each Query Response is considered to express a constraint upon the universe of possible solutions, with the actual solution being intersection of the constrained spaces. An equivalent Solution Space is therefore:

[{{?Student => Jones, ?Major=>CS},
{?Major => Math, ?Student =>Doe}}],

Part of the goal of our algorithm is to eventually reduce the Solution Space to a single Query Response like this last one.

Fig. 1 describes the top-level algorithm for answering a query. A query is answered by a process of progressively restricting the SolutionSpace by adding variable bindings (in the form of Query Responses). The initial space with no bindings ❶ represents a completely unconstrained SolutionSpace. The input query consists of a sequence of query patterns.

We repeatedly estimate the response size for the remaining query patterns ❷, and choose the most restrictive pattern ❸ to be considered next. We solve the chosen pattern by backward chaining ❹, and then merge the variable bindings obtained from backward chaining into the SolutionSpace ❺ via the restrictTo function, which performs a (possibly deferred) join as described later in this section.

When all query patterns have been processed, if the Solution Space has not been reduced to a single Query Response, we perform a final join of these variable bindings into single one variable binding that contains all the variables involved in all the query patterns ❻. The finalJoin function is described in more detail later in this section.

The estimation of response sizes in ❷ can be carried out by a combination of 1) exploiting the fact that each pattern represents that application of a predicate with known domain and range types. If these positions in the triple are occupied by variables, we can check to see if the variable is already bound in our SolutionSpace and to how many values it is bound. If it is unbound, we can estimate the size of the domain (or range) type, 2) accumulating statistics on typical response sizes for previously encountered patterns involving that predicate. The effective mixture of these sources of information is a subject for future work.

For example, suppose there are 10,000 students, 500 courses, 50 faculty members and 10 departments in the knowledgebase. For the query pattern (?S takesCourse ?C), the domain of takesCourse is Student, while the range of takesCourse is Course. An estimate of the numbers of triples matching the pattern (?S takesCourse ?C) might be 100,000

```
QueryResponse answerAQuery(query: Query)
{
   // Set up initial SolutionSpace
   SolutionSpace solutionSpace = empty; ❶

   // Repeatedly reduce SolutionSpace by
applying
   //   the most restrictive pattern
   while (unexplored patterns remain
          in the query) {
       computeEstimatesOfReponseSize
         (unexplored patterns); ❷
       QueryPattern p = unexplored pattern
           with smallest estimate; ❸

       // Restrict SolutionSpace via
       //   exploration of p

      QueryResponse answerToP =
         BackwardChain(p); ❹
      solutionSpace.restrictTo
        (answerToP); ❺
    }
    return solutionSpace.finalJoin();❻
}
```

Figure 1. Answering a Query.

TABLE I. EXAMPLE QUERY 1

| Clause # | QueryPattern | Query Response |
|---|---|---|
| 1 | ?S1 takesCourse ?C1 | $\{(?S1=>s_i,?C1=>c_i)\}_{i=1..100,000}$ |
| 2 | ?S1 takesCourse ?C2 | $\{(?S1=>s_j, ?C2=>c_j)\}_{j=1..100,000}$ |
| 3 | ?C1 taughtBy fac1 | $\{(?C1=>c_j)\}_{j=1..3}$ |
| 4 | ?C2 taughtBy fac1 | $\{(?C2=>c_j)\}_{j=1..3}$ |

if the average number of courses a student has taken is ten, although the number of possibilities is 500,000.

By using a greedy ordering ❸ of the patterns within a query, we hope to reduce the average size of the SolutionSpaces. For example, suppose that we were interested in listing all cases where any student took multiple courses from a specific faculty member. We can represent this query as the sequence of the patterns in Table I. These clauses are shown with their estimated result sizes indicated in the subscripts. The sizes used in this example are based on one of our LUBM benchmark [20] prototypes .

To illustrate the effect of the greedy ordering, let us assume first that the patterns are processed in the order given. A trace of the answerAQuery algorithm, showing one row for each iteration of the main loop is shown in Table II. The worst case in terms of storage size and in terms of the size of the sets being joined is at the join of clause 2, when the join of two sets of size 100,000 yields 1,000,000 tuples.

Now, consider the effect of applying the same patterns in ascending order of estimated size, shown in Table III. The worst case in terms of storage size and in terms of the size of the sets being joined is at the final addition of clause 2, when a set of size 100,000 is joined with a set of 270. Compared to Table II, the reduction in space requirements and in time required to perform the join would be about an order of magnitude.

TABLE II. TRACE OF JOIN OF CLAUSES IN THE ORDER GIVEN

| Clause Being Joined | Resulting SolutionSpace |
|---|---|
| (initial) | [ ] |
| 1 | $[\{(?S1=>s_i, ?C1=>c_i)\}_{i=1..100,000}]$ |
| 2 | $[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c_i)\}_{i=1..1,000,000}]$ (based on an average of 10 courses / student) |
| 3 | $[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c_i)\}_{i=1..900}]$ (Joining this clause discards courses taught by other faculty.) |
| 4 | $[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c_i)\}_{i=1..60}]$ |

TABLE III. TRACE OF JOIN OF CLAUSES IN ASCENDING ORDER OF ESTIMATED SIZE

| Clause Being Joined | Resulting SolutionSpace |
|---|---|
| (initial) | [ ] |
| 3 | $[[\{(?C1=>c_i)\}_{i=1..3}]$ |
| 4 | $[\{(?C1=>c_i, ?C2=>c_j)\}_{i=1..3, j=1..3}]$ |
| 1 | $[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c'_j)\}_{i=1..270}]$ |
| 2 | $[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c_i)\}_{i=1..60}]$ |

```
void SolutionSpace::restrictTo
(QueryResponse newbinding)
{
   for each element oldBinding
      in solutionSpace
   {
      if (newbinding shares variables
          with oldbinding){ ❶
         bool merged = join(newBinding,
                     oldBinding, false);❷
         if (merged) {
            remove oldBinding from
              solutionSpace;
         }
      }
   }
   add newBinding to solutionSpace;
}
```

Figure 2. Restricting a SolutionSpace.

The output from the backward chaining reasoner will be a query response. These must be merged into the current SolutionSpace as a set of additional restrictions. Fig. 2 shows how this is done.

Each binding already in the SolutionSpace ❶ that shares at least one variable with the new binding ❷ is applied to the new binding, updating the new binding so that its domain is the union of the sets of variables in the old and new bindings and the specific functions represent the constrained cross-product (join) of the two. Any such old bindings so joined to the new one can then be discarded.

The join function at ❷ returns the joined QueryResponse as an update of its first parameter. The join operation is carried out as a hash join [22] with an average complexity $O(n_1+n_2+m)$ where the $n_i$ are the number of tuples in the two input sets and $m$ is the number of tuples in the joined output.

The third (boolean) parameter of the join call indicates whether the join is forced (true) or optional (false), and the boolean return value indicates whether an optional join was actually carried out. Our intent is to experiment in future versions with a dynamic decision to defer optional joins if a partial calculation of the join reveals that the output will far exceed the size of the inputs, in hopes that a later query clause may significantly restrict the tuples that need to participate in this join.

As noted earlier, our interpretation of the SolutionSpace is that it denotes a set of potential bindings to variables, represented as the join of an arbitrary number of QueryResponses. The actual computation of the join can be deferred, either because of a dynamic size-based criterion as just described, or because of the requirement at ❶ that joins be carried out immediately only if the input QueryResponses share at least one variable. In the absence of any such sharing, a join would always result in an output size as long as the products of its input sizes. Deferring such joins can help reduce the size of the SolutionSpace and, as a consequence, the cost of subsequent joins.

For example, suppose that we were processing a different example query to determine which mathematics courses are

TABLE IV. EXAMPLE QUERY 2

| Clause | QueryPattern | Query Response |
|---|---|---|
| 1 | (?S1 takesCourse ?C1) | {(?S1=>s$_j$,?C1=>c$_j$)}$_{j=1..100,000}$ |
| 2 | (?S1 memberOf CSDept) | {(?S1=>s$_j$)}$_{j=1..1,000}$ |
| 3 | (?C1 taughtby ?F1) | {(?C1=>c$_j$, ?F1=>f$_j$)}$_{j=1..1,500}$ |
| 4 | (?F1 worksFor MathDept) | {(?F1=>f$_i$)}$_{i=1..50}$ |

taken by computer science majors, represented as the sequence of the following QueryPatterns, shown with their estimated sizes in Table IV.

To illustrate the effect of deferring joins on responses that do not share variables, even with the greedy ordering discussed earlier, suppose, first, that we perform all joins immediately. Assuming the greedy ordering that we have already advocated, the trace of the answerAQuery algorithm is shown in Table V.

In the prototype from which this example is taken, the Math department teaches 150 different courses and there are 1,000 students in the CS Dept. Consequently, the merge of clause 3 (1,500 tuples) with the SolutionSpace then containing 50,000 tuples yields considerably fewer tuples than the product of the two input sizes. The worst step in this trace is the final join, between sets of size 100,000 and 150,000.

But consider that the join of clause 2 in that trace was between sets that shared no variables. If we defer such joins, then the first SolutionSpace would be retained “as is”. The resulting trace is shown in Table VI.

The subsequent addition of clause 3 results in an immediate join with only one of the responses in the solution space. The response involving ?S1 remains deferred, as it shares no variables with the remaining clauses in the SolutionSpace. The worst join performed would have been between sets of size 100,000 and 150, a considerable improvement over the non-deferred case.

TABLE V. TRACE OF JOIN OF CLAUSES IN ASCENDING ORDER OF ESTIMATED SIZE

| Clause Being Joined | Resulting SolutionSpace |
|---|---|
| (initial) | [] |
| 4 | [{(?F1=>f$_i$)}$_{i=1..50}$] |
| 2 | [{(?F1=>f$_i$, ?S1=>s$_i$)}$_{i=1..50,000}$] |
| 3 | [{(?F1=>f$_i$, ?S1=>s$_i$, ?C1=>c$_i$)}$_{i=1..150,000}$] |
| 1 | [{(?F1=>f$_i$, ?S1=>s$_i$, ?C1=>c$_i$)}$_{i=1..1,000}$] |

TABLE VI. TRACE OF JOIN OF CLAUSES WITH DEFERRED JOINS

| Clause Being Joined | Resulting SolutionSpace |
|---|---|
| (initial) | [] |
| 4 | [{(?F1=>f$_i$)}$_{i=1..50}$] |
| 2 | [{(?F1=>f$_i$)}$_{i=1..50}$,{(?S1=>s$_j$)}$_{j=1..1,000}$] |
| 3 | [{(?F1=>f$_i$, ?C1=>c$_i$)}$_{i=1..150}$ , {(?S1=>s$_j$)}$_{j=1..1,000}$] |
| 1 | [{(?F1=>f$_i$, ?S1=>s$_i$, ?C1=>c$_i$)}$_{i=1..1,000}$] |

```
QueryResponseSolutionSpace::finalJoin ()
{
  sort the bindings in this solution
   space into ascending order by
   number of tuples;  ❶

   QueryResponse result = first of the
     sorted bindings;
   for each remaining binding b
     in solutionSpace {
      join (result, b, true);  ❷
   }
   return result;
}
```

Figure 3. Final Join.

When all clauses of the original query have been processed (Fig. 1 ❻), we may have deferred several joins because they involved unrelated variables or because they appeared to lead to a combinatorial explosion on their first attempt. The finalJoin function shown in Fig. 3 is tasked with reducing the internal SolutionSpace to a single QueryResponse, carrying out any join operations that were deferred by the earlier restrictTo calls. In many ways, finalJoin is a recap of the answerAQuery and restrictTo functions, with two important differences:

- Although we still employ a greedy ordering ❶ to reduce the join sizes, there is no need for estimated sizes because the actual sizes of the input QueryResponses are known.
- There is no longer an option to defer joins between QueryResponses that share no variables. All joins must be performed in this final stage❷ and so the “forced” parameter to the optional join function is set to true.

## IV. EVALUATION

In this section we compare our answerAQuery algorithm of Fig. 1 against an existing system, Jena, that also answers queries via a combination of an in-memory backward chaining reasoner with basic knowledge base retrievals.

The comparison was carried out using LUBM benchmarks representing a knowledge base describing a single university and a collection of 10 universities. Prior to the application of any reasoning, these benchmarks contained 100,839 and 1,272,871 triples, respectively.

We evaluated these using a set of 14 queries taken from LUBM [20]. These queries involve properties associated with the LUBM university-world ontology, with none of the custom properties/rules whose support is actually our end goal (as discussed in [23]). Answering these queries requires, in general, reasoning over rules associated with both RDFS and OWL semantics, though some queries can be answered purely on the basis of the RDFS rules.

Table VII compares our algorithm to the Jena system using a pure backward chaining reasoner. Our comparison will focus on response time, as our optimization algorithm should be neutral with respect to result accuracy, offering no more and no less accuracy than is provided by the interposed reasoner.

TABLE VII COMPARISON AGAINST JENA WITH BACKWARD CHAINING

| **LUBM:** | **1 University, 100,839 triples** | | | | **10 Universities, 1,272,871 triples** | | | |
|---|---|---|---|---|---|---|---|---|
| | ***answerAQuery*** | | ***Jena Backwd*** | | ***answerAQuery*** | | ***Jena Backwd*** | |
| | ***response time*** | ***result size*** | ***response time*** | ***result size*** | ***response time*** | ***result size*** | ***response time*** | ***result size*** |
| Query1 | 0.20 | 4 | 0.32 | 4 | 0.43 | 4 | 0.86 | 4 |
| Query2 | 0.50 | 0 | 130 | 0 | 2.1 | 28 | n/a | n/a |
| Query3 | 0.026 | 6 | 0.038 | 6 | 0.031 | 6 | 1.5 | 6 |
| Query4 | 0.52 | 34 | 0.021 | 34 | 1.1 | 34 | 0.41 | 34 |
| Query5 | 0.098 | 719 | 0.19 | 678 | 0.042 | 719 | 1.0 | 678 |
| Query6 | 0.43 | 7,790 | 0.49 | 6,463 | 1.9 | 99,566 | 3.2 | 82,507 |
| Query7 | 0.29 | 67 | 45 | 61 | 2.2 | 67 | 8,100 | 61 |
| Query8 | 0.77 | 7,790 | 0.91 | 6,463 | 3.7 | 7,790 | 52 | 6,463 |
| Query9 | 0.36 | 208 | n/a | n/a | 2.5 | 2,540 | n/a | n/a |
| Query10 | 0.18 | 4 | 0.54 | 0 | 1.8 | 4 | 1.4 | 0 |
| Query11 | 0.24 | 224 | 0.011 | 0 | 0.18 | 224 | 0.032 | 0 |
| Query12 | 0.23 | 15 | 0.0020 | 0 | 0.33 | 15 | 0.016 | 0 |
| Query13 | 0.025 | 1 | 0.37 | 0 | 0.21 | 33 | 0.89 | 0 |
| Query14 | 0.024 | 5,916 | 0.58 | 5,916 | 0.18 | 75,547 | 2.6 | 75,547 |

As a practical matter, however, Jena’s system cannot process all of the rules in the OWL semantics rule set, and was therefore run with a simpler rule set describing only the RDFS semantics. This discrepancy accounts for the differences in result size (# of tuples) for several queries. Result sizes in the table are expressed as the number of tuples returned by the query and response times are given in seconds. An entry of n/a means that the query processing had not completed (after 1 hour).

Despite employing the larger and more complicated rule set, our algorithm generally ran faster than Jena, sometimes by multiple orders of magnitude. The exceptions to this behavior are limited to queries with very small result set sizes or queries 10-13, which rely upon OWL semantics and so could not be answered correctly by Jena. In two queries (2 and 9), Jena timed out.

Jena also has a hybrid mode that combines backward chaining with some forward-style materialization. Table VIII shows a comparison of our algorithm with a pure backward chaining reasoner against the Jena hybrid mode. Again, an n/a entry indicates that the query processing had not completed within an hour, except in one case (query 8 in the 10 Universities benchmark) in which Jena failed due to exhausted memory space.

The times here tend to be someone closer, but the Jena system has even more difficulties returning any answer at all when working with the larger benchmark. Given that the difference between this and the prior table is that, in this case, some rules have already been materialized by Jena to yield, presumably, longer lists of tuples, steps taken to avoid possible combinatorial explosion in the resulting joins would be increasingly critical.

TABLE VIII. COMPARISON AGAINST JENA WITH WITH HYBRID REASONER

| **LUBM** | **1 University, 100,839 triples** | | | | **10 Universities, 1,272,871 triples** | | | |
|---|---|---|---|---|---|---|---|---|
| | ***answerAQuery*** | | ***Jena Hybrid*** | | ***answerAQuery*** | | ***Jena Hybrid*** | |
| | ***response time*** | ***result size*** | ***response time*** | ***result size*** | ***response time*** | ***result size*** | ***response time*** | ***result size*** |
| Query1 | 0.20 | 4 | 0.37 | 4 | 0.43 | 4 | 0.93 | 4 |
| Query2 | 0.50 | 0 | 1,400 | 0 | 2.1 | 28 | n/a | n/a |
| Query3 | 0.026 | 6 | 0.050 | 6 | 0.031 | 6 | 1.5 | 6 |
| Query4 | 0.52 | 34 | 0.025 | 34 | 1.1 | 34 | 0.55 | 34 |
| Query5 | 0.098 | 719 | 0.029 | 719 | 0.042 | 719 | 2.7 | 719 |
| Query6 | 0.43 | 7,790 | 0.43 | 6,463 | 1.9 | 99,566 | 3.7 | 82,507 |
| Query7 | 0.29 | 67 | 38 | 61 | 2.2 | 67 | n/a | n/a |
| Query8 | 0.77 | 7,790 | 2.3 | 6,463 | 3.7 | 7,790 | n/a | n/a |
| Query9 | 0.36 | 208 | n/a | n/a | 2.5 | 2,540 | n/a | n/a |
| Query10 | 0.18 | 4 | 0.62 | 0 | 1.8 | 4 | 1.6 | 0 |
| Query11 | 0.24 | 224 | 0.0010 | 0 | 0.18 | 224 | 0.08 | 0 |
| Query12 | 0.23 | 15 | 0.0010 | 0 | 0.33 | 15 | 0.016 | 0 |
| Query13 | 0.025 | 1 | 0.62 | 0 | 0.21 | 33 | 1.2 | 0 |
| Query14 | 0.024 | 5,916 | 0.72 | 5,916 | 0.18 | 75,547 | 2.5 | 75,547 |

## V. CONCLUSION /FUTURE WORK

As knowledge bases proliferate on the Web, it becomes more plausible to add reasoning services to support more general queries than simple retrievals. In this paper, we have addressed a key issue of the large amount of information in a semantic web of data about science research. Scale in itself is not really the issue. Problems arise when we wish to reason about the large amount of data and when the information changes rapidly. In this paper, we report on our efforts to use backward-chaining reasoners to accommodate the changing knowledge base. We developed a query-optimization algorithm that will work with a reasoner interposed between the knowledge base and the query interpreter. We performed experiments, comparing our implementation with traditional backward-chaining reasoners and found, on the one hand, that our implementation could handle much larger knowledge bases and, on the other hand, could work with more complete rule sets (including all of the OWL rules). When both reasoners produced the same results our implementation was never worse and in most cases significantly faster (in some cases by orders of magnitude).

In a future paper we will address the issue of being able to scale the knowledgebase to the level forward-chaining reasoners can handle. Preliminary results indicate that we can scale up to real world situations such as 6 Million triples. Optimizing the backward-chaining reasoner, together with the query-optimization reported on in this paper, will allow us to actually outperform forward-chaining reasoners in scenarios where the knowledge base is subject to frequent change

Finally, we will be working on creating a hybrid reasoner that will combine the two reasoners. We will need to be able to identify the impact on the knowledgebase specific changes have. How does the reasoner know if a fact is in the 'trusted' region or needs to be re-inferenced? How do we find facts which are revoked by a change? If we succeed we can then apply Backward reasoning only to incremental changes and periodically will do a full materialization (which does scale to billions of triples)on which we can do simple look-ups.

## VI. REFERENCES

[1] S. J. Russell and P. Norvig, Artificial intelligence: a modern approach., 1st ed. , Prentice hall, 1995, pp. 265–275.

[2] The Apache Software Foundation, Apache Jena, 2013 [retrieved: March, 2013 ], available from: http://jena.apache.org/.

[3] Microsoft, Microsoft Academic Search, 2013 [retrieved: March, 2013 ], available from: http://academic.research.microsoft.com/.

[4] Z. Nie, Y. Zhang, J. Wen and W. Ma, "Object-level ranking: bringing order to web objects", Proceedings of the 14th international World Wide Web conference, ACM Press, Chiba, Japan, May 2005, pp. 567–574, doi:10.1145/1060745.1060828.

[5] A. Doan, et al., "Community information management", IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases, vol. 29, iss. 1, March 2006, pp. 64–72.

[6] J. Tang, et al., "Arnetminer: Extraction and mining of academic social networks", Proceedings of the Fourteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'2008), ACM Press, Las Vegas, USA, August 2008, pp. 990–998, doi: 10.1145/1401890.1401891.

[7] C. Bizer, et al., "DBpedia-A crystallization point for the Web of Data", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 7, iss. 3, Sep. 2009, pp. 154–165, doi: 10.1016/j.websem.2009.07.002.

[8] F. Suchanek, G. Kasneci, G. Weikum, "Yago: A large ontology from wikipedia and wordnet", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 6, iss.3, Sep. 2008, pp.203–217, doi: 10.1016/j.websem.2008.06.001

[9] B. Aleman-Meza, F. Hakimpour, I. Arpinar and A. Sheth, "SwetoDblp ontology of Computer Science publications", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 5, iss. 3, Sep. 2007, pp. 151–155, doi: 10.1016/j.websem.2007.03.001.

[10] H. Glaser, I. Millard, and A. Jaffri, "Rkbexplorer. com: a knowledge driven infrastructure for linked data providers", European Semantic Web Conference, Springer-Verlag, Tenerife, Spain, Jun. 2008, pp. 797–801.

[11] A. Kiryakov, D. Ognyanov and D. Manov, "OWLIM–a pragmatic semantic repository for OWL", Proceedings of the 2005 international conference on Web Information Systems Engineering(WISE'05), Springer-Verlag , New York, USA, Nov. 2005, pp. 182-192, doi: 10.1007/11581116_19.

[12] Oracle Corporation, Oracle Database 11g R2, 2013 [retrieved: March, 2013], Available from: http://www.oracle.com/technetwork /database/enterprise-edition/overview/index.html

[13] O. Erling, I. Mikhailov, "RDF Support in the Virtuoso DBMS", Networked Knowledge-Networked Media, vol. 221, 2009, pp. 7-24, doi: 10.1007/978-3-642-02184-8_2.

[14] Y.E. Ioannidis, "Query optimization", ACM Computing Surveys (CSUR), vol. 28, iss. 1, March 1996, pp. 121-123, doi: 10.1145/234313.234367.

[15] Semanticweb.org, SPARQL endpoint, 2011 [retrieved: March, 2013], available from: http://semanticweb.org/wiki /SPARQL_endpoint.

[16] W3C, SparqlEndpoints. 2013 [retrieved: March 2013], Available from: http://www.w3.org/wiki/SparqlEndpoints.

[17] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer and D. Reynolds, "SPARQL basic graph pattern optimization using selectivity estimation", Proceedings of the 17th international conference on World Wide Web, ACM Press, Beijing, China, April 2008, pp. 595–604, doi: 10.1145/1367497.1367578.

[18] O. Hartig and R. Heese "The SPARQL query graph model for query optimization", Proceedings of the 4th European conference on the Semantic Web: Research and Applications (ESWC '07), Springer-Verlag, Innsbruck, Austria, Jun. 2007, pp. 564–578, doi: 10.1007/978-3-540-72667-8_40.

[19] W. Le, "Scalable multi-query optimization for SPARQL", 2012 IEEE 28th International Conference on Data Engineering (ICDE), IEEE Press, Washington, DC, April 2012, pp. 666–677, doi: 10.1109/ICDE.2012.37.

[20] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, iss. 2-3, Oct. 2005, pp.158–182, doi: 10.1016/j.websem.2005.06.005.

[21] L. Ma, et al., "Towards a complete OWL ontology benchmark". Proceedings of the 3rd European conference on The Semantic Web: research and applications(ESWC'06), Springer-Verlag, June 2006, p. 125–139, doi: 10.1007/11762256_12.

[22] M. Kitsuregawa, H. Tanaka and T. Moto-Oka, "Application of hash to data base machine and its architecture". New Generation Computing, vol. 1, iss.1, March 1983, pp. 63–74.

[23] H. Shi, K. Maly, S. Zeil and M. Zubair, "Comparison of Ontology Reasoning Systems Using Custom Rules", International Conference on Web Intelligence, Mining and Semantics, ACM Press, Sogndal, Norway, May 2011, doi: 10.1145/1988688.1988708.