A Scalable Backward Chaining-based Reasoner for a Semantic Web

Hui Shi
Department of Management and Information Sciences
University of Southern Indiana
Evansville, USA
hshi@cs.odu.edu

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

Abstract — **In this paper we consider knowledge bases that organize information using ontologies. Specifically, we investigate reasoning over a semantic web where the underlying knowledge base covers linked data about science research that are being harvested from the Web and are supplemented and edited by community members. In the semantic web over which we want to reason, frequent changes occur in the underlying knowledge base, and less frequent changes occur in the underlying ontology or the rule set that governs the reasoning. 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. We present an algorithm for dynamic query optimization in such an architecture. We also introduce new optimization techniques to the backward-chaining algorithm. We show that these techniques together with the query-optimization reported on earlier, will allow us to actually outperform forward-chaining reasoners in scenarios where the knowledge base is subject to frequent change. Finally, we analyze the impact of these techniques on a large knowledge base that requires external storage.**

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 some of these qualitative descriptors such as "groundbreaking research in data mining" may come to be accepted as meaningful if they represent a consensus of an appropriate subset of the community at large. However, even in the absence of such sharing, we believe the expressiveness of user-defined qualitative descriptors is highly desirable.

The system implied by these queries is an example of a semantic web service where the underlying knowledge base 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 knowledge base, which would be updated continuously [1].

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 [2].

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 knowledge base 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 knowledge base 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 [1].

In this paper, we will also address the issue of being able to scale the knowledge base beyond the level standard backward-chaining reasoners can handle. We shall introduce new optimization techniques to a backward-chaining algorithm and shall show that these techniques, together with query-optimization, will allow us to actually outperform forward-chaining reasoners in scenarios where the knowledge base is subject to frequent change.

Finally, we explore the challenges posed by scaling the knowledge base to a point where external storage is required. This raises issues about the middleware that handles external storage, how to optimize the amount of data and what data are to be moved to internal storage.

In Section II, we provide background material on the semantic web, reasoning, and database querying. Section III gives the overall query-optimization algorithm for answering a query. In Section IV, we report on experiments comparing our new algorithm with a commonly used backward chaining algorithm. Section V introduces the optimized backward-chaining algorithm and Section VI provides details on the new techniques we have introduced to optimize performance. A preliminary evaluation of these techniques on a smaller scale, using in-memory storage, is reported in a separate paper [3]. In Section VII, we describe the issues raised when scaling to an externally stored knowledge base, evaluate the performance of our query optimization and reasoner optimizations in that context, and perform an overall comparison with different data base implementations.

II. Related Work

A number of projects (e.g., Libra [4][5], Cimple [6], and Arnetminer [7]) 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 [8-11] 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 [2]. 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 [12] and Oracle 11g [13], 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 [14], for example, implements a mixture of forward-chaining and backward-chaining. Jena [15] supports three ways of inferencing: forward-chaining, limited backward-chaining and a hybrid of these two methods.

In conventional database management systems, query optimization [16] 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 [17][18], rely on a variety of techniques for estimating the cost of query components, including selectivity estimations [19], graph optimization [20], and cost models [21]. These techniques assume a fully materialized knowledge base.

Benchmarks evaluate and compare the performances of different reasoning systems. The Lehigh University Benchmark (LUBM) [22] 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.

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

```
QueryResponseanswerAQuery(query: Query)
{
   // Set up initial SolutionSpace
   SolutionSpacesolutionSpace = 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
     QueryResponseanswerToP =
        BackwardChain(p); ❹
     solutionSpace.restrictTo (
        answerToP); ❺
    }
    return solutionSpace.finalJoin();❻
}
```

Figure 1. Answering a Query.

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 SolutionSpace 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 knowledge base. 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 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 [22] 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 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	$\{(?S1=>s_i, ?C2=>c_j)\}_{j=1..100,000}$
3	?C1 taughtBy fac1	$\{(?C1=>c_j)\}_{j=1..3}$
4	?C2taughtBy fac1	$\{(?C2=>c_j)\}_{j=1..3}$

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	$[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c_i)\}_{i=1..60}]$

The output from the backward chaining reasoner will be a query response. These must be merged into the currentSolutionSpace 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 [23] 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

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	$[\{(?C1=>c_i, ?C2=>c_i)\}_{i=1..3, j=1..3}]$
1	$[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c'_i)\}_{i=1..270}]$
2	$[\{(?S1=>s_i, ?C1=>c_i, ?C2=>c_i)\}_{i=1..60}]$

```
void SolutionSpace::restrictTo (QueryRe-
sponsenewbinding)
{
   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.

cost of subsequent joins.

When all clauses of the original query have been processed (Fig. 1❻), 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 ❶to 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.

For example, suppose that we were processing a different example query to determine which mathematics courses are taken by computer science majors, represented as the sequence of the following QueryPatterns, shown with their estimated sizes in Table IV.

```
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.

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)	$\{(?F1=>f_i)\}_{i=1..50}$

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.

IV. EVALUATION OF QUERY OPTIMIZATION

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 two LUBM benchmarks consisting of one knowledge base describing a single university and another describing 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 [22]. These queries involve properties associated with the LUBM university-world ontology, with none of the custom properties/rules whose support is actually our end

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	$[\{(?F1=>f_i, ?S_1=>s_i, ?C1=>c_i)\}_{i=1..150,000}]$
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	$[\{(?F1=>f_i, ?S1=>s_i, ?C1=>c_i)\}_{i=1..1,000}]$

goal (as discussed in [3]). 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 focuses 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.

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 ruleset 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 trend 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.

V. OPTIMIZED BACKWARD CHAINING ALGORITHM

When the knowledge base is dynamic, backward chaining is a suitable choice for ontology reasoning. However, as the size of the knowledge base increases, standard backward chaining strategies [2][15] do not scale well for ontology reasoning. In this section, first, we discuss issues some backward chaining methods expose for ontology reasoning. Second, we present our backward chaining algorithm that introduces new optimization techniques as well as addresses the known issues.

A. Issues

1. Guaranteed Termination: Backward chaining is usually implemented by employing a depth-first search strategy. Unless methods are used to prevent it, the depth-first search could go into an infinite loop. For example, in our rule set, we have rules that involve each other when proving their heads:

rule1: (?P owl:inverseOf ?Q) -> (?Q owl:inverseOf ?P)
rule2;(?P owl:inverseOf ?Q), (?X ?P ?Y) -> (?Y ?Q ?X)

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

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

In order to prove body clause ?P owl:inverseOf ?Q in rule1, we need to prove the body of rule2 first, because the head of rule2 matches body clause ?P owl:inverseOf ?Q. In order to prove the first body clause ?P owl:inverseOf ?Q in rule2, we also need to prove the body clause ?P owl: inverseOf ?Q in rule1, because the head of rule1 matches body clause ?P owl:inverseOf ?Q.

Even in cases where depth-first search terminates, the performance may suffer due to time spent exploring, in depth, branches that ultimately do not lead to a proof.

We shall use the OLDT [24] method to avoid infinite recursion and will introduce optimizations aimed at further performance improvement in Section VI.C.

2. The owl:sameAs Problem: The built-in OWL property owl:sameAs links two equivalent individuals. An owl:sameAs triple indicates that two linked individuals have the same "identity" [25]. An example of a rule in the OWL-Horst rule set that involves the owl:sameAs relations is the rule: "(?x owl:sameAs ?y) (?x ?p ?z) -> (?y ?p ?z)".

Consider a triple, which has m owl:sameAs equivalents of its subject, n owl:sameAs equivalents of its predicate, and k owl:sameAs equivalents of its object, Then m*n*k triples would be derivable from that triple.

Reasoning with the owl:sameAs relation can result in a multiplication of the number of instances of variables during backward-chaining and expanded patterns in the result. As long as that triple is in the result set, all of its equivalents would be in the result set as well. This adds cost to the reasoning process in both time and space.

B. The Algorithm

The purpose of this algorithm is to generate a query response for a given query pattern based on a specific rule set. We shall use the following terminology.

A VariableBinding is a substitution of values for a set of variables.

A RuleSet is a set of rules for interpretation by the reasoning system. This can include RDFS Rules [26], Horst rules [27] and custom rules [28] that are used for ontology reasoning. For example,

[rdfs1: (?x ?p ?y) -> (?p rdf:type rdf:Property)].

The main algorithm calls the function BackwardChaining, which finds a set of triples that can be unified with pattern with bindings varList, any bindings to variables appearing in headClause from the head of applied rule, bodylist that are reserved for solving the recursive problem. Given a Goal and corresponding matched triples, a QueryResponse is created and returned in the end.

Our optimized BackwardChaining algorithm, described in Fig. 4, is based on conventional backward chaining algorithms [2]. The solutionList is a partial list of solutions already found for a goal.

For a goal that has already been resolved, we simply get the results from solutionList. For a goal that has not been resolved yet, we will seek a resolution by applying the rules. We initially search in the knowledge base to find triples that match the goal (triples in which the subject, predicate and object are compatible with the query pattern). Then, we find rules with heads that match the input pattern. For each such rule we attempt to prove it by proving the body clauses (new goals) subject to bindings from already-resolved goals from the same body. The process of proving one rule is explained below. The method of "OLDT" [24] is adopted to solve the non-termination issue we mentioned in Section VI.C. Finally, we apply any "same as" relations to candidateTriples to solve the owl:sameAs problem. During this process of "SameAsTripleSearch", we add all equivalent triples to the existing results to produce complete results.

Fig. 5 shows how to prove one rule, which is a step in Fig. 4. The heart of the algorithm is the loop through the clauses of a rule body, attempting to prove each clause. Some form of selection function is implied that selects the next unproven clause for consideration on each iteration. Traditionally, this would be left-to-right as the clauses are written in the rule. Instead, we order the body clauses by the number of free variables. The rationale for this ordering will be discussed in the following Section VI. A.

```
BackwardChaining(pattern,headClause,bodylist,level,varList)
{
   if (pattern not in solutionList){
      candidateTriples+= matches to pattern that found in knowledge base;
      solutionList+= mapping from pattern to candidateTriples;
      relatedRules = rules with matching heads to pattern that found in ruleList;
      realizedRules = all the rules in relatedRules with substitute variables from pattern;
      backupvarList = back up clone of varList;
      for (each oneRule in realizedRules){
         if(attemptToProveRule(oneRule, varList, level)){
            resultList= unify(headClause, varList);
            candidateTriples+= resultList;
         }
         oldCandidateTriples = triples in mappings to headClause from solutionList;
         if ( oldCandidateTriples not contain candidateTriples){
            update solutionList with candidateTriples;
            if(UpdateafterUnificationofHead(headClause, resultList))
            {
                newCandidateTriples = triples in mappings to headClause from solutionList;
                candidateTriples+= newCandidateTriples;
            }
         }
       }
    }
    else /* if (solutionList.contains(pattern)) */
    {
       candidateTriples+= triples in mappings to pattern from solutionList;
       Add reasoning context, including head and bodyRest to lookupList;
    }
    SameAsTripleSearch(candidateTriples);
    return candidateTriples;
}
```

Figure 4. Process of BackwardChaining.

The process of proving one goal (a body clause from a rule) is given in Fig. 6. Before we prove the body clauses (new goals) in each rule, the value of a calculated dynamic threshold decides whether we perform the substitution or not. We substitute the free variables in the body clause with bindings from previously resolved goals from the same body. The step helps to improve the reasoning efficiency in terms of response time and scalability and will be discussed in Section VI.B. We call the BackwardChaining function to find a set of triples that can be unified with body clause (new goal) with substituted variables. Bindings will also be updated gradually following the proof of body clauses.

```
attemptToProveRule(oneRule,varList,level)
{
   body = rule body of oneRule;
   sort body by ascending number of free
      variables;
   head = rule head of oneRule;
   for (each bodyClause in body)
   {
      canBeProven =
        attemptToProveBodyClause (
          bodyClause, body, head,
          varList, level);
       if (!canBeProven) break;
   }
   return canBeProven;
}
```

Figure 5. Process of proving one rule.

VI. OPTIMIZATION DETAILS & DISCUSSION

There are four optimizations that have been introduced in our algorithm for backward chaining. These optimizations are: 1) the implementation of the selection function, which implements the ordering the body clauses in one rule by the number of free variables, 2) the upgraded substitute function, which implements the substitution of the free variables in the body clauses in one rule based on calculating a threshold that switches resolution methods, 3) the application of OLDT and 4) solving of the owl:sameAs problem. Of these, optimization 1 is an adaptation of techniques employed in other reasoning contexts [29][30] and optimizations 3 and 4 have appeared in [24, 31] whereas techniques 2 are new. We will describe the implementation details of these optimizations below. A preliminary evaluation of these techniques is reported in a separate paper. [3] A more extensive evaluation is reported here in Section VII.

A. Ordered Selection Function

The body of a rule consists of a conjunction of multiple clauses. Traditional SLD (Selective Linear Definite) clause resolution systems such as Prolog would normally attempt these in left-to-right order, but, logically, we are free to attempt them in any order.

```
attemptToProveBodyClause(goal, body,
head, varList, level)
{
  canBeProven = true;
  dthreshold = Calculate dynamic
    threshold;
  patternList = get unified patterns by
    replacing variables in bodyClause
    from varList for current level with
    calculated dthreshold;
  for(each unifiedPattern in
      patternList ) {
     if(!unifiedPattern.isGround()) {
         bodyRest = unprocessedPartOf(
           body, goal);
        triplesFromResolution+=
           BackwardChaining(
             unifiedPattern, head,
             bodyRest, level+1,
             varList);
     }
     else if(unifiedPattern.isGround()) {
        if (knowledgeBase contains
             unifiedPattern){
           triplesFromResolution+=
             unifiedPattern;
        }
     }
  }
  if(triplesFromResolution.size()>0) {
     update_varList with varList,
       triplesFromResolution, goal, and
       level;
     if (varList==null) {
        canBeProven = false;
     }
  }
  else{
      canBeProven = false;
  }
  return canBeProven;
}
```

Figure 6. Process of proving one goal.

We expect that given a rule under proof, ordering the body clauses into ascending order by the number of free variables will help to decrease the reasoning time. For example, let us resolve the goal "?y rdf:type Student", and consider the rule:

[rdfs3: (?x ?p ?y) (?p rdfs:range ?c) -> (?y rdf:type ?c)]

The goal "?y rdf:type Student" matches the head of rule "?y rdf:type ?c", and ?c is unified with Student.

If we select body clause "?x ?p ?y" to prove first, it will yield more than 5 million (using LUBM(40) [22]) instances of clauses. The proof of body clause "?x ?p ?y" in backward chaining would take up to hours. Result bindings of "?p" will be propagated to the next body clause "?p rdfs:range ?c" to yield new clauses (p1 rdfs:range Student), (p2 rdfs:range Student), …, (p32 rdfs:range Student), and then a separate proof would be attempted for each of these specialized forms.

If we select body clause "?p rdfs:range Student" (?c is unified with Student) to prove first, it will yield zero (using LUBM(40)) instances of clauses. The proof of body clause "?p rdfs:range Student" would take up to seconds. No result bindings would be propagated to body clause "?x ?p ?y". The process of proof terminates.

The body clause "?p rdfs:range ?c" has one free variable ?p while the body clause "?x ?p ?y" has three free variables. It is reasonable to prove body clause with fewer free variables first, and then propagate the result bindings to ?p to next body clause "?x ?p ?y". Mostly, goals with fewer free variables cost less time to be resolved than goals with more free variables, since fewer free variables means more bindings and body clauses with fewer free variables will match fewer triples.

B. Switching between Binding Propagation and Free Variable Resolution

Binding propagation and free variable resolution are two modes of for dealing with conjunctions of multiple goals. We claim that dynamic selection of these two modes during the reasoning process will increase the efficiency in terms of response time and scalability.

These modes differ in how they handle shared variables in successive clauses encountered while attempting to prove the body of a rule. Suppose that we have a rule body containing clauses (?x p1 ?y) and (?y p2 ?z) [other patterns of common variables are, of course, also possible] and that we have already proven that the first clause can be satisfied using value pairs $\{(x_1, y_1), (x_2,y_2),\ldots(x_n,y_n)\}$.

In the binding propagation mode, the bindings from the earlier solutions are substituted into the upcoming clause to yield multiple instances of that clause as goals for subsequent proof. In the example given above, the value pairs from the proof of the first clause would be applied to the second clause to yield new clauses (y1 p2 ?z), (y2 p2 ?z), …, (yn p2 ?z), and then a separate proof would be attempted for each of these specialized forms. Any (y,z) pairs obtained from these proofs would then be joined to the (x,y) pairs from the first clause.

In the free variable resolution mode, a single proof is attempted of the upcoming clause in its original form, with no restriction upon the free variables in that clause. In the example above, a single proof would be attempted of (?y p2 ?z), yielding a set of pairs $\{(y_n, z_1), (y_{n+1},z_2),\ldots(x_{n+k},z_k)\}$. The join of this with the set $\{(x_1, y_1), (x_2,y_2),\ldots(x_n,y_n)\}$ would then be computed to describe the common solution of both body clauses.

The binding propagation mode is used for most backward chaining systems [15]. There is a direct tradeoff of multiple proofs of narrower goals in binding propagation against a single proof of a more general goal in free variable resolution. As the number of tuples that solve the first body clause grows, the number of new specialized forms of the subsequent clauses will grow, leading to higher time and space cost overall. If the number of tuples from the earlier clauses is large enough, free variable resolution mode will be more efficient. (In the experimental results in Section VII, we will

demonstrate that neither mode is uniformly faster across all problems.)

Following is an example (using LUBM(40)) showing one common way of handling shared variables between body clauses.

Suppose we have an earlier body clause 1: "?y type Course" and a subsequent body clause 2: "?x takesCourse ?y". These two clauses have the common variable ?y. In our experiments, it took 1.749 seconds to prove body clause 1 while it took an average of 0.235 seconds to prove body clause 2 for a given value of ?y from the proof of body clause 1. However, there were 86,361 students satisfying variable ?x, which means it would take 0.235 *86,361=20,295 seconds to finish proof of 86,361 new clauses after applying value pairs from the proof of body clause 1. 20,295 seconds is not acceptable as query response time. We need to address this problem to improve reasoning efficiency in terms of response time and scalability.

We propose to dynamically switch between modes based upon the size of the partial solutions obtained so far. Let n denote the number of solutions that satisfy an already proven clause. Let t denote threshold used to dynamically select between modes. If $n \leqslant t$, then the binding propagation mode will be selected. If $n>t$, then the free variable resolution mode will be selected. The larger the threshold is, the more likely binding propagation mode will be selected.

Suppose that we have a rule body containing clauses (a1 p1 b1) (a2 p2 b2). Let (a1 p1 b1) be the first clause, and (a2 b2 c2) be the second clause. ai, bi and ci ($i \in [1,2]$) could be free variable or concrete value. Assume that there is at least one common variable between two clauses.

In the binding propagation mode, the value pairs from the proof of the first clause would be applied to the second clause to yield new clauses ($a2_1$ $p2_1$ $b2_1$), ($a2_2$ $p2_2$ $b2_2$), …, ($a2_n$ $p2_n$ $c2_n$), and then a separate proof would be attempted for each of these specialized forms. Any value sets obtained from these proofs would then be joined to the value sets from the first clause. Let $join_1$ denote the time spent on the joint operations. Let $proof_1^i$ denote the time of proving first clause with i free variables and $proof_2^j$ be the average time of proving new specialized form with j free variables. ($i \in [1,3]$, $j \in [0,2]$)

In the free variable resolution mode, a single proof is attempted of the upcoming clause in its original form, with no restriction upon the free variables in that clause. A single proof would be attempted of (a2 p2 b2), yielding a set of value sets. The join of the value sets yielded from the first clause and the values sets yielded from the second clause would then be computed to describe the common solution of both body clauses. Let $join_2$ denote the time spent on the joint operations. Let $proof_3^k$ denote the time of proving second clause with k free variables. ($k \in [1,3]$)

Determining t is critical to switching between two modes. Let us compare the time spent on binding propagation mode and free variable resolution mode to determine t. Binding propagation is favored when

$$proof_1^i + proof_2^j * n + join_1 < proof_1^i + proof_3^k + join_2$$

Isolating the term involving n,

$$proof_2^j * n < proof_3^k + join_2 - join_1$$

$join_1$ is less than or equal to $join_2$, because the value sets from the second clause in the binding propagation mode have already been filtered by the value sets from the first clause first. The join operations in binding propagation mode are therefore a subset of the join operations in free variable resolution mode. Let t be the largest integer value such that

$$proof_2^j * t < proof_3^k$$

then

$$proof_2^j * t <= proof_2^j * n < proof_3^k + join_2 - join_1$$

We conclude that:

$$t = floor(proof_3^k / proof_2^j) \quad (1)$$

Formula (1) provides thus a method for calculating the threshold t that determines when to employ binding propagation. In that formula, k denotes the number of free variables in the second clause (a2 p2 b2), j denotes the number of free variables of the new specialized forms ($a2_1$ $p2_1$ $b2_1$), ($a2_2$ $p2_2$ $b2_2$), ($a2_n$ $p2_n$ $c2_n$) of the second clause with ($k \in [1,3]$, $j \in [0,2]$). The specialized form of the second clause has one or two less free variables than the original form. Hence, the possible combinations of (k,j) are {(3,2), (3,1), (2,1), (2,0), (1,0)}.

To estimate $proof_3^k$ and $proof_2^j$, we record the time spent on proving goals with different numbers of free variables. We separately keep a record of the number of goals that have one free variable, two free variables and three free variables after we start calling our optimized backwardChaining algorithm. We also record the time spent on proving these goals. After we have recorded a sufficient number of proof times (experiments will give us an insight into what constitutes a 'sufficient' number), we compute the average time spent on goals with k free variables and j free variables, respectively, to obtain an estimate of $proof_3^k$ and proof2j.

In order to adopt accurate threshold to help improve the efficiency, we apply different thresholds to different situations with corresponding number of free variable set (k,j).

We assign the initial value to t from previous experiments in a particular knowledge base/query environment if they exist or zero otherwise.

We update the threshold several times when answering a particular query. The threshold will change as different queries are being answered. For each query, we will call the optimized backward chaining algorithm recursively several times. Each call of backwardChaining is given a specific goal as an input. During the running of backwardChaining, the average time of proving a goal as a function of the number of free variables will be updated after a goal has been proven. During the running of backwardChaining, every time before making selection between two modes the estimate threshold is updated before making the decision.

C. How to Avoid Repetition and Non-Termination

Given RDFS Rules [26], Horst rules [27] and custom rules [28] in the rule set and queries for answering, backward chaining for ontology reasoning may hit the same goals for several times. Some body clauses such as ?a rdfs:subClassOf ?b and ?x rdfs:subPropertyOf ?y appear in

multiple rules in Horst rule set that is used in many reasoning systems. During the process of answering a given query, these rules containing the same body clauses might be necessary to be proved to answer the query. During the process of answering a given query, some rules may be repeatedly called for more than one time, leading to proving the same body clause like ?a rdfs:subClassOf ?b more than one time. Within the process of answering one query, such a repetition decreases the efficiency in terms of response time. Backward chaining with memorization will help to avoid repetition.

Backward chaining is implemented in logic programming [32] by SLD resolution [33]. When we apply conventional backward chaining process to ontology reasoning, it has the same non-termination problem as SLD resolution does. During the proving process, the rule body needs to be satisfied to prove the goal. In some cases, the rule body requires proving goals that have the same property as the goal, resulting possibly in an infinite loop unless steps are taken to ensure termination.

For example, [rdfs8: (?a rdfs:subClassOf ?b), (?b rdfs:subClassOf ?c) -> (?a rdfs:subClassOf ?c)] is one rule in the RDFS rule set used for ontology reasoning. When we apply standard backward chaining to ontology reasoning, proving the head (?a rdfs:subClassOf ?c) requires proving of the body (?a rdfs:subClassOf ?b) and (?b rdfs:subClassOf ?c). This loop will be infinite without applying any techniques.

We use an adaptation of the OLDT algorithm to solve this non-termination problem. The OLDT algorithm is an extension of the SLD-resolution [33] with a left to right computation rule. OLDT maintains a solution table and lookup table to solve the recursion problem.

D. owl:sameAs Optimization

The "owl:sameAs" relation poses a problem [31] for almost all the reasoning systems including forward chaining. In our reasoning system, we first pre-compute all possible owl:sameAs pairs and save them to a sameAs table. Second, we select a representative node to represent an equivalence class of owl:sameAs URIs. Third, we replace the equivalence class of owl:sameAs URIs with the representative node. At last, if users want to return all the identical results, we populate the query response using the sameAs table by replacing the representative node with the URIs in the equivalence class.

As we described in Section V, reasoning with the owl:sameAs relation can result in a multiplication of the number of instances of variables during backward-chaining and expanded patterns in the result. As long as that triple is in the result set, all of the members in its equivalence class would be in the result set as well. This adds cost to the reasoning process in both time and space. The optimization that applies pre-computation and selects a representative node improves the performance in terms of time and space.

This optimization is a novel adaptation of owl:sameAs optimization in forward chaining reasoning system, such as OWLIM-SE [34] and Oracle [13], to backward chaining reasoning systems.

VII. BACKWARD CHAINING WITH EXTERNALLY STORED KNOWLEDGE BASE

In Section IV and in our earlier experiments assessing the effectiveness of our optimized reasoner [3], all our experiments were performed 'in-memory', which limited the study to a knowledge base of less than 10 Million triples.

In this section, we switch to implementations that use external storage for the knowledge base. We consider Jena SDB [35], Jena TDB [36] and OWLIM-SE [34]. We extend our study based on a knowledge base of more than 100 Million triples.

The employment of external storage introduces new factors and has implications on how to improve the scalability of our backward chaining reasoner. First, any optimization technique needs to balance the number of accesses to data and the size of the retrieved data against the size of in-memory cache and its use. Second, the algorithm has to take now into account that it will take longer to access a triple (or a set of triples) due to having to perform I/O. In-memory reasoners typically have a 'model' of the knowledge base in which they store the facts and an API to access them. When an external storage is used they would provide transparent connections from the model to the external databases that would allow the reasoner to use the same API for accessing the model. This leads to a third factor effecting the scalability and performance of the reasoner: the middleware that realizes the transparent linking.

Jena SDB provides persistent triple stores using relational databases. An SQL database is required for the storage and query of triples for SDB. In this paper, we used MySQL and PostgreSQL as the relational database for SDB. Jena TDB is claimed as a more scalable and faster triple store than SDB [35].A special Jena adapter permits access to OWLIM-SE repositories [34]. Reasoners can access all three storage systems via a common Jena API.

A. Preliminary Analysis

We begin by exploring the relative impact on overall performance of the three major components of the backward chaining reasoner, the middleware, and the storage system itself. The purpose of this analysis is to determine how much time we can save by improving any one of these subsystems in isolation.

We employed Jena SDB + MySQL as the external storage for our backward chaining reasoner in the experiment, evaluating the query response time of 14 queries from LUMB [22] using LUBM(30).

A single function in our backward chaining algorithm implementation is responsible for all data retrievals from the triple store. We refer to this function as "the Data-retrieval function" in the remainder of this section. Data-retrieval function in this paper. We recorded the clock time T_f and CPU time t_f spent within the Data-retrieval function and in the whole query processing (T_{tot} and t_{tot}, respectively) in Table IX.

The portion of the CPU and clock times spent I answering the query but not spent in the Data-retrieval function is attributable to the backward chaining reasoner:

TABLE IX CLOCK TIME, CPU TIME AND I/O TIME FROM EXPERIMENTS WITH JENA SDB USING LUBM (30)

	Total Clock time, T_{tot}	Total CPU Time, t_{tot}	Clock time in Data-retrieval function, T_f	CPU time in I/O function, t_f
Query1	1405.00	951.00	920.00	546.00
Query2	9631.00	6084.00	5058.00	2293.00
Query3	203.00	78.00	109.00	31.00
Query4	35354.00	8096.00	31140.00	5070.00
Query5	173.00	78.00	94.00	15.00
Query6	23744.00	7035.00	19984.00	3712.00
Query7	24058.00	9984.00	18659.00	6333.00
Query8	28694.00	11029.00	22680.00	5896.00
Query9	29598.00	11700.00	23899.00	6988.00
Query 10	18612.00	6630.00	15040.00	3572.00
Query 11	3636.00	561.00	2964.00	124.00
Query 12	7567.00	1903.00	5226.00	405.00
Query 13	187.00	46.00	95.00	0.00
Query 14	1873.00	811.00	1451.00	452.00

$T_{bw} = T_{tot} - T_f$
$T_{bw} = t_{tot} - t_f$

The clock time observed during the Data-retrieval function includes actual input operations on the underlying triple store, together with the CPU-intensive manipulation of the input data by the middleware layer. Assuming that the ratio, $\rho = t_{tot}/T_{tot}$, of CPU time to clock time observed over the processing of an entire query would remain approximately constant during the middleware CPU, we were able to estimate the portion of the Data-retrieval function clock time that was attributable to the middleware:

$T_{mid} = \rho \ t_{mid}$

and can attribute the remaining clock time as the actual time spent doing I/O:

$T_{IO} = T_f - T_{mid}$

Then we can estimate a minimal clock time to answer the query, assuming 100% CPU utilization, as

$T_{min} = t_{bw} + \rho \ T_{mid} + T_{IO}$

Table X shows the values of these estimates, together the percentage of that value attributable to each of the three components. In Table X, the percentage of time spent in I/O operations ranges from 22% to 75%, a considerable variation. This might be because some retrievals from triple store retrieve huge numbers of triples while others are far more focused and process much less data.

TABLE X ESTIMATED I/O TIME AND IDEAL PERCENTAGES FROM EXPERIMENTS WITH JENA SDB USING LUBM (30)

	Min possible clock time to answer a query, T_{min}	% of T_{min} spent in I/O	% of Tmin spent in BW chaining	% of T_{min} time spent in middleware
Query1	1217.15	0.22	0.33	0.45
Query2	8376.00	0.27	0.45	0.27
Query3	125.00	0.38	0.38	0.25
Query4	32175.53	0.75	0.09	0.16
Query5	153.19	0.49	0.41	0.10
Query6	22818.84	0.69	0.15	0.16
Query7	19277.93	0.48	0.19	0.33
Query8	26801.04	0.59	0.19	0.22
Query9	27147.26	0.57	0.17	0.26
Query 10	17497.60	0.62	0.17	0.20
Query 11	3334.32	0.83	0.13	0.04
Query 12	6496.09	0.71	0.23	0.06
Query 13	141.00	0.67	0.33	0.00
Query 14	1730.68	0.53	0.21	0.26

The percentage of the time devoted to the middleware ranges from 0% to 44%, with an average around 20%, indicating that the triple storage layer adds a significant component of CPU time. Our backward chaining code running on top of that accounts for 13 to 45% of minimal processing time, and the average is 25%.

These percentages are surprisingly balanced, suggesting that improvements to any one of the three major components of the system can have only modest effect on the total time. Dramatic improvements will be possible only by improvement in all three areas. One possible avenue of exploration is changes to the reasoner that would not only speed up the reasoner but would affect the number and size of requests for input from the underlying store. Indirectly, at least, several of the optimizations we have proposed in Section VI could have such an effect. Caching, an effect not explored in this experiment, could also have a major impact across all three areas.

B. Evaluation of the Optimization techniques

In this section, we examine the impact of the two major optimizations proposed in Section VI.

1) Ordered Selection Function

We have proposed replacing the traditional left-to-right processing of clauses within rule bodies by an ordering by ascending number of free variables.

Table XI compares our backward chaining algorithm with our clause selection based on free variable count to the traditional left-to-right selection on a relatively small knowledge base (100,839 triples) LUBM(1) [22] stored in Jena TDB. Traditional left-to-right selection has been used in Jena [15] and Prolog [32]. Backward chaining with the ordered selection function yields considerably smaller query response times for all the queries than left-to-right. The I/O time of accessing the external triple storage magnifies the problem of left-to-right selection compared to [3] because the knowledge base is in external triple storage TDB now.

The difference becomes even more dramatic for a larger knowledge base (1,272,871 triples), LUBM(10) stored in Jena TDB, as shown in Table XII. With left-to-right selection, we are unable to answer any query within 30 minutes, and out-of-memory errors occur for almost half of the queries. The I/O time of accessing the external triple storage magnifies the problem of left-to-right selection compared to [3] because the knowledge base is in external triple storage TDB now.

2) Switching between Binding Propagation and Free Variable Resolution

Binding propagation and free variable resolution are two modes of for dealing with conjunctions of multiple goals. We have proposed dynamic selection of these two modes during the reasoning process to increase the efficiency in terms of response time and scalability.

We compare our backward chaining algorithm with three different modes of resolving goals on LUBM(10) stored in Jena TDB in Table XIII. The first mode uses dynamic selection between binding propagation mode and free variable resolution mode. The second mode uses binding propagation mode only. The third mode uses free variable resolution mode only.

Table XIII shows that neither binding propagation mode nor free variable resolution mode is uniformly better than the other on all cases. From query 1 to query 5 and query 13, dynamic mode performs almost same as binding propagation mode. From query 6 to query 10, dynamic mode performs dramatically better than binding propagation mode with much less query response time. For query 11, query 12 and query 14, dynamic mode performs better than binding propagation mode with less query response time.

For query1, query3 and query 14 only, dynamic mode performs almost same as free variable resolution mode. For the other queries, dynamic mode performs dramatically better than free variable resolution mode with much less query response time. The query response times of query6 to query10 are less by orders of magnitude when running our algorithm with the dynamic selection mode in comparison compared to running with binding propagation mode only and free variable resolution mode only. In all cases the optimized version finishes faster than the better of the other two versions. Overall, the results in Table XIII confirm the advantage of dynamically selecting between propagation modes. The I/O time of accessing the external triple storage magnifies the problem of binding propagation mode only and free variable resolution mode only compared to [3] be-

TABLE XI. EVALUATION OF CLAUSE SELECTION OPTIMIZATION ON LUBM(1) USING TDB AS EXTERNAL STORAGE

	Time (ms), Ordered	Time (ms), Left-to right	Result Size (triples)
Query1	296	>6.0*105	4
Query2	811	>6.0*105	0
Query3	46	>6.0*105	6
Query4	1419	>6.0*105	34
Query5	31	>6.0*105	719
Query6	265	>6.0*105	7,790
Query7	234	>6.0*105	67
Query8	483	>6.0*105	7,790
Query9	202	>6.0*105	208
Query10	156	>6.0*105	4
Query11	218	>6.0*105	224
Query12	202	>6.0*105	15
Query13	15	>6.0*105	1
Query14	31	>6.0*105	5,916

TABLE XII. EVALUATION OF CLAUSE SELECTION OPTIMIZATION ON LUBM(10) USING TDB AS EXTERNAL STORAGE

	Time (ms), Ordered	Time (ms), Left-to right	Result Size (triples)
Query1	1045	OutOfMemoryError: Java heap space	4
Query2	2433	>2.0*106	28
Query3	31	>2.0*106	6
Query4	3744	>2.0*106	34
Query5	15	>2.0*106	719
Query6	1435	OutOfMemoryError	99,566
Query7	1903	OutOfMemoryError	67
Query8	2106	OutOfMemoryError	7,790
Query9	1918	OutOfMemoryError	2,540
Query10	1138	OutOfMemoryError	4
Query11	140	>2.0*106	224
Query12	358	>2.0*106	15
Query13	15	>2.0*106	33
Query14	187	>2.0*106	75,547

TABLE XIII . EVALUATION OF DYNAMIC SELECTION VERSUS BINDING PROPAGATION AND FREE VARIABLE MODES ON LUBM(10) USING TDB AS EXTERNAL STORAGE

	Time (ms), Dynamic selection	Time (ms), Binding propagation only	Time (ms), Free variable resolution only
Query1	1045	904	904
Query2	2433	2683	26535
Query3	31	15	15
Query4	3744	4149	41605
Query5	15	15	2244810
Query6	1435	>6.0*105	20514
Query7	1903	>6.0*105	20763
Query8	2106	>6.0*105	42831
Query9	1918	>6.0*105	21512
Query10	1138	>6.0*105	19921
Query11	140	904	19094
Query12	358	1435	41745
Query13	15	31	24117
Query14	187	1154	187

cause the knowledge base are in external triple storage TDB now. The selection of the threshold in dynamic mode would be affected by the employment of external storage and affect the number of accesses to store.

C. Storage System Impact

To explore the effect of switching the underlying storage manager, we compared three external storage employed in our optimized backward chaining reasoner on I/O time. For all 14 queries from LUBM, the three storage managers SDB, TDB and OWLIM-SE, all have same number of accesses (calls to the Data-retrieval function) to the underlying store.

Based on this observation, we show in Table XIV the I/O time per access for SDB, TDB and OWLIM-SE using LUBM(50). The I/O time per store access of SDB is dramatically longer than both TDB and OWLIM-SE through all 14 queries in LUBM. From query 1 to 5 and query 13, the I/O time per store access of TDB is slightly longer than OWLIM-SE. For the other queries, TDB has shorter I/O time per store access. In general, TDB and OWLIM-SE have the similar performance in terms of I/O time.

D. Overall Performance

Finally, we consider the overall performance of our optimized backward chaining reasoner when based upon each of the three storage managers.

In order to compare the general performance of three triple store when employed in our optimized backward chaining reasoner, for all 14 queries from LUBM, we perform a comparison among SDB, TDB and OWLIM-SE on query response time using LUBM(50) in Table XV .

In Table XV, for LUBM (50), from query 1 to query 3 and query 6, OWLIM-SE has the fastest response time. Jena SDB + PostgreSQL performs fastest only for query 4, because the I/O time of Jena SDB is the longest out of three stores. For the rest of the queries, Jena TDB is fastest.

In Table XVI, we show a similar comparison of TDB and OWLIM-SE on query response time using LUBM(100). SDB was omitted from this comparison because the loading time of SDB is prohibitively long.

TABLE XIV .COMPARISON AMONG SDB, TDB AND OWLIM-SE AS EXTERNAL STORAGE ON I/O TIME PER STORE ACCESS

	LUBM(50)			
Number of facts (triples)	6,890,640			
	Time (ms), SDB+ PostgreSQL	***Time (ms), TDB***	***Time (ms), OWLIM-SE***	***#of Number of access to store***
Query1	41.42	2.32	0.70	132
Query2	50.76	0.48	0.35	353
Query3	1.63	0.42	0.28	65
Query4	82.38	0.38	0.14	455
Query5	1.57	0.36	0.20	81
Query6	298.74	0.67	5.12	153
Query7	237.69	0.13	0.52	286
Query8	72.24	0.07	0.43	917
Query9	221.45	0.02	0.17	351
Query10	223.33	0.07	0.14	218
Query11	2.08	0.05	0.12	616
Query12	2.07	0.03	0.10	2792
Query13	1.28	0.21	0.13	86
Query14	111.76	0.03	0.22	67

TABLE XV.COMPARISON BETWEEN SDB, TDB AND OWLIM-SE AS EXTERNAL STORAGE ON QUERY RESPONSE TIME

	LUBM(50)		
Number of facts (triples)	6,890,640		
	Clock Time		
	Time (ms), SDB+PostgreSQL	***Time (ms), TDB***	***Time (ms), OWLIM-SE***
Query1	6430	13440	3549
Query2	24960	36102	17046
Query3	406	58	61
Query4	46400	71298	45680
Query5	533	78	156
Query6	59144	32590	30470
Query7	83799	34580	45527
Query8	85563	48307	53013
Query9	95992	34583	49566
Query10	63100	20191	27916
Query11	3466	528	876
Query12	16253	2403	3199
Query13	374	39	37
Query14	8581	4731	5364

TABLE XVI.COMPARISON BETWEEN SDB, TDB AND OWLIM-SE AS EXTERNAL STORAGE ON QUERY RESPONSE TIME

	LUBM(100)	
Number of facts (triples)	13,405,677	
	Time (ms), TDB	***Time (ms), OWLIM-SE***
Query1	2652	5085
Query2	13884	29657
Query3	31	46
Query4	49109	82664
Query5	46	78
Query6	26020	51277
Query7	39873	76752
Query8	58609	98343
Query9	46925	85456
Query10	26894	52821
Query11	452	826
Query12	920	1716
Query13	15	31
Query14	7222	11263

In Table XVI, for LUBM(50), Jena TDB has better performance through all 14 queries. In general, our optimized backward chaining reasoner and external storage Jena TDB has the best performance especially when the size of the knowledge base increases.

VIII. Conclusion and 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).

The analysis of reasoning over a large knowledge base that requires external storage has shown that no one component (backward chaining, I/O, middleware) dominates performance and thus improvements to any one of the three major components of the system will have only modest effect on the total time.

We have also addressed the issue of being able to scale the knowledge base 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 allows us to actually outperform forward-chaining reasoners in scenarios where the knowledge base is subject to frequent change.

Although 6 million triples remains a modest size for a knowledge base, we believe that the key performance limitation is associated with the number of triples that are being brought into memory as intermediate results during the reasoning for a specific query. In [37] we tie the use of reasoning to a concept of "trust" reflecting changes made to the knowledge base since its last instantiation. Trust can be exploited to decide what goals arising during evaluation of a query require reasoning and what can be resolved by immediate lookup. The net effect is that considerably larger knowledge bases can be handled by limiting the scope of backward chaining to portions of the knowledge base untrusted due to recent changes.

Assessing the impact of using external storage on the individual optimization techniques produced in both of the cases we analyzed the same result. Having an external triple store magnified the effect of our optimization techniques. When we analyzed storage access we found that SDB was significantly slower than TDB and OWLIM-SE. The latter two had about the same performance. As the size of the knowledge base kept increasing the advantage of using Jena TDB with our optimized backward-chaining algorithm became more pronounced.

We will explore in future work ways to minimize in our backward chaining algorithms the number and size of requests for input from the underlying store and to employ caching techniques.

References

[1] H. Shi, K. Maly, and S. Zeil, "Query optimization in cooperation with an ontological reasoning service," The Fifth International Conferences on Advanced Service Computing (SERVICE COMPUTATION 2013), IARIA XPS Press, May-Jun. 2013, pp. 26-32.

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

[3] H. Shi, K. Maly, and S. Zeil, "Optimized backward chaining reasoning system for a semantic web," Proc. The Fourth International Conference on Web Intelligence, Mining and Semantics (WIMS'14), ACM Press, June 2014.

[4] Microsoft. *Microsoft Academic Search*. [Online]. Available from: http://academic.research.microsoft.com/ 2014.02.25

[5] Z. Nie, Y. Zhang, J. Wen, and W. Ma, "Object-level ranking: bringing order to web objects," The 14th international World Wide Web conference (WWW2005), ACM Press, May 2005, pp. 567–574, doi:10.1145/1060745.1060828.

[6] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen., "Community information management," IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases, vol. 29, iss. 1, pp. 64–72, Mar. 2006.

[7] J.Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su, "Arnetminer: extraction and mining of academic social networks," Proc. Fourteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'2008), ACMPress, Aug. 2008, pp. 990–998, doi:10.1145/1401890.1402008.

[8] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, and C. Becker, "DBpedia-a crystallization point for the Web of Data," Web Semantics: Science, Services and Agents on the World Wide Web, vol. 7, iss. 3, pp. 154–165, Sep. 2009, doi:10.1016/j.websem.2009.07.002.

[9] F. Suchanek, G. Kasneci, and 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, pp.203–217, Sep. 2008, doi:10.1016/j.websem.2008.06.001.

[10] 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, pp. 151–155, Sep. 2007, doi:10.1016/j.websem.2007.03.001.

[11] H. Glaser, I. Millard, and A. Jaffri, "Rkbexplorer. com: a knowledge driven infrastructure for linked data providers," The Semantic Web: Research and Applications, vol. 5021, pp. 797–801, Jun. 2008, doi:10.1007/978-3-540-68234-9_61.

[12] A. Kiryakov, D. Ognyanov, and D. Manov, "OWLIM–a pragmatic semantic repository for OWL," Proc. 6th international conference on Web Information Systems Engineering (WISE'05), Springer-Verlag, pp. 182-192, Nov. 2005, doi:10.1007/11581116_19.

[13] Oracle Corporation. 2013. *Oracle Database 11g R2*. [Online]. Available from: http://www.oracle.com/technetwork/database/database-technologies/express-edition/overview/ 2014.02.25

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

[15] The Apache Software Foundation. *Apache Jena*. [Online]. Available from: http://jena.apache.org/ 2014.02.25

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

[17] Semanticweb.org. *SPARQL endpoint*. [Online]. Available from:http://semanticweb.org/wiki/SPARQL_endpoint 2014.02.25

[18] W3C. *SparqlEndpoints*. [Online]. Available from: http://www.w3.org/wiki/SparqlEndpoints 2014.02.25

[19] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D.Reynolds, "SPARQL basic graph pattern optimization using selectivity estimation," The 17th international conference on World Wide Web (WWW 2008), ACM Press, pp. 595–604, Apr. 2008, doi:10.1145/1367497.1367578.

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

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

[22] 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, pp.158–182, Oct. 2005, doi:10.1016/j.websem.2005.06.005.

[23] 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, pp. 63–74, Mar. 1983, doi:10.1007/BF03037022.

[24] H. Tamaki and T. Sato, "OLD resolution with tabulation," Proc. Third international conference on logic programming, Springer, pp. 84-98, July 1986, doi:10.1007/3-540-16492-8_66.

[25] W3C. *OWL web ontology language reference*. [Online]. Available from: http://www.w3.org/TR/owl-ref/ 2014.02.25

[26] P. Hayes and B. McBride. *RDF semantics*. [Online]. Available from: http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ 2014.02.25

[27] H. Horst, "Combining RDF and part of OWL with rules: Semantics, decidability, complexity," Proc. 4th International Semantic Web Conference (ISWC 2005), Springer, pp. 668-684, Nov. 2005, doi:10.1007/11574620_48.

[28] 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, May 2011, doi: 10.1145/1988688.1988708.

[29] K. Marriott and P. J. Stuckey, Programming with constraints: an introduction. Cambridge: MIT press, 1998.

[30] J. Santos and S. Muggleton, "When does it pay off to use sophisticated entailment engines in ILP?," in Inductive Logic Programming, P. Frasconi and F. A. Lisi, Eds. Heidelberg: Springer, pp. 214-221, 2011.

[31] Ontotext. *Owl-sameAs-optimization*. [Online]. Available from: http://www.ontotext.com/owlim/owl-sameas-optimisation 2014.02.25

[32] J. Lloyd, "Foundations of Logic Programming," 2nd extend ed.. Springer-Verlag: Berlin, 1987.

[33] R. Kowalski and D. Kuehner, "Linear resolution with selection function," Artificial Intelligence, vol. 2, iss. 3, pp. 227-260, 1972, doi: 10.1016/0004-3702(71)90012-9.

[34] Ontotext. *OWLIM-SE*. [Online]. Available from: http://owlim.ontotext.com/display/OWLIMv43/OWLIM-SE 2014.02.25

[35] The Apache Software Foundation. *SDB - persistent triple stores using relational databases*. [Online]. Available from: http://jena.apache.org/documentation/sdb/ 2014.02.25

[36] The Apache Software Foundation. *TDB*. [Online]. Available from: http://jena.apache.org/documentation/tdb/ 2014.02.25

[37] H. Shi, K. Maly, and S. Zeil, "Trust and hybrid reasoning for ontological knowledge bases," Proc. the companion publication of the 23rd international conference on World wide web companion (WWW Companion '14), International World Wide Web Conferences Steering Committee, pp. 1189-1194, April 2014, doi: 10.1145/2567948.2579033.