RC24562 (W0805-126) May 20, 2008
Computer Science

IBM Research Report

Efficient Reasoning on Large SHIN Aboxes in Relational Databases

Julian Dolby[1], Achille Fokoue[1], Aditya Kalyanpur[1], Li Ma[2], Chintan Patel[3], Edith Schonberg[1], Kavitha Srinivas[1], Xing Zhi Sun[2]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[2]IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100094
P.R.China

[3]Columbia University Medical Center

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient reasoning on large SHIN Aboxes in relational databases

Julian Dolby[1], Achille Fokoue[1], Aditya Kalyanpur[1], Li Ma[2], Chintan Patel[3], Edith Schonberg[1], Kavitha Srinivas[1], and Xingzhi Sun[2]

[1] IBM Watson Research Center,P.O.Box 704, Yorktown Heights, NY 10598, USA
dolby, achille, adityakal, ediths, ksrinivs@us.ibm.com
[2] IBM China Research Lab, Beijing 100094, China
malli, sunxingz@cn.ibm.com
[3] Columbia University Medical Center
chintan.patel@dbmi.columbia.edu

Abstract. As applications based on semantic web technologies enter the mainstream, there is a need to provide highly efficient ontology reasoning over large Aboxes. A common approach to achieving scalability is to build reasoners for DL subsets (e.g., the EL family of languages, DL-Lite, DLP, or OWL-Prime). However, the proliferation of DL subsets runs counter to standardization efforts. In this paper, we present a hybrid approach which combines a fast, incomplete reasoning algorithm with a slower complete reasoning algorithm to handle the more expressive features of DL. Our approach works for $\mathcal{SHIN}$. We demonstrate the effectiveness of this approach on large datasets (30-60 million assertions), where we show that performance on this hybrid approach provides significant performance gains (an average of 15 mins per query compared to 100 mins) without sacrificing completeness or expressivity.
keywords: Reasoning, Description Logic, Ontology.

1 Introduction

As applications based on semantic web technologies enter the mainstream, there is a need to provide highly efficient ontology reasoning over large Aboxes. However, it is well known that description logic (DL) reasoning is intractable in the worst case. A common approach to achieving scalability is to define useful subsets of DL that have polynomial-time complexity, and build reasoners for these subsets (e.g., the EL family of languages, DL-Lite, DLP, or OWL-Prime). While this approach satisfies the needs of some applications, there are many scenarios where the reasoning requirements exceed the expressivity of these subsets of DL, and where completeness is a requirement. Furthermore, the proliferation of DL subsets runs counter to standardization efforts.

In this paper, we present a hybrid approach that combines a fast, incomplete reasoning algorithm with a slower complete reasoning algorithm to handle the more expressive features of DL. Our approach works for $\mathcal{SHIN}$ (OWL-DL without nominals or datatypes). An interesting feature of this technique is that

any sound and incomplete algorithm may be used in the first phase to quickly find as many solutions as possible to the query. The key novelty in the approach is a mechanism to incorporate these solutions into a slower, complete reasoning algorithm for $\mathcal{SHIN}$, providing much better performance characteristics overall, without sacrificing completeness or expressivity. This approach can be described as self-adjusting, since the reasoner dynamically defaults to the expensive complete algorithm only when deeper inferencing is actually required. On large datasets (30-60 million assertions), this hybrid approach provides significant performance gains (an average of 15 mins per query on the 60 million dataset compared to 100 mins) without sacrificing completeness or expressivity.

At its core, this hybrid approach builds on the summarization and refinement techniques we described earlier to perform sound and complete reasoning on large Aboxes in relational databases [1] [2]. Briefly, this technique applies a standard tableaux algorithm on a *summary Abox* $\mathcal{A}'$ rather than the original Abox $\mathcal{A}$ to answer queries. A summary Abox is created by aggregating individuals which are members of the same concepts, so when any given individual is tested in the summary Abox, all individuals mapped to the summary individual are effectively tested at the same time. For a tested individual s in $\mathcal{A}'$, if the summary is found to be consistent, then we know that all individuals mapped to that summary individual s are not solutions. But if the summary is found to be inconsistent, it is possible that either (a) a subset of individuals mapped to the summarized individual s are instances of the query or (b) the inconsistency is a spurious effect of the summarization. We determine the answer through *refinement*, which selectively expands the summary Abox to make it more precise. Refinement is an iterative process that partitions the set of individuals mapped to a single summary individual based on the common edges they have in the original Abox, and remaps each partition to a new summary individual. The iteration ends when either the expanded summary is consistent, or it can be shown that all individuals mapped to the tested summary individual are solutions. Significantly, convergence on the solution is based only on the structure of the refined summary, without testing individuals in $\mathcal{A}$. In practice, the scalability of this algorithm is limited by the number of refinement steps that are needed. Refinement is performed by database join operations, which become expensive when the database is large.

The advantage of our hybrid approach is that it can incorporate any sound and incomplete reasoning algorithm into this summarization and refinement process to provide efficient, complete, and yet highly scalable reasoning over large Aboxes. The key insight is that the solutions from the sound and incomplete reasoner can be used as a partitioning function for refinement instead of partitioning based on common edges, as described in our earlier work. This effectively removes the obvious solutions from the summary Abox. If the sound and incomplete reasoning algorithm finds all solutions, there will be no solutions left in the summary Abox after this first refinement, so the algorithm will converge very quickly. Any remaining inconsistencies are spurious, and can be resolved in one or a few refinement steps. If the sound and incomplete algorithm finds only some

of the solutions, then the refinement process will find the rest of the solutions with fewer refinement steps. If there are no solutions found by the sound and incomplete reasoner, the number of refinement steps is the same.

Our key contributions in this paper are as follows: (a) we describe a fast, sound but incomplete algorithm, which finds most but not all solutions in typical usage scenarios, (b) we describe how to incorporate these solutions into a sound, complete algorithm for large Aboxes, and (c) we demonstrate its effectiveness in providing performance gains (from 100 minutes per query to 15 minutes per query) on expressive Aboxes with 60 million assertions.

2 Background

Query answering in expressive DLs can be reduced to consistency detection. For instance, assume that we want to find all instances of the concept C. To answer this query, each individual a is tested by adding the assertion $a : \neg C$ to the Abox, and checking the new Abox for consistency. If the Abox is inconsistent, then a is an instance of C. For large Aboxes, this approach will clearly not scale. Therefore, in our previous work [2], we modify this approach to perform tableau reasoning on a summarized version of the Abox rather than the original Abox. Formally, an Abox $\mathcal{A}'$ is a summary Abox of a $\mathcal{SHIN}$ Abox $\mathcal{A}$ if there is a mapping function $\mathbf{f}$ that satisfies the following constraints[4]:

(1) if $a : C \in \mathcal{A}$ then $\mathbf{f}(a) : C \in \mathcal{A}'$
(2) if $R(a, b) \in \mathcal{A}$ then $R(\mathbf{f}(a), \mathbf{f}(b)) \in \mathcal{A}'$
(3) if $a \neq b \in \mathcal{A}$ then $\mathbf{f}(a) \neq \mathbf{f}(b) \in \mathcal{A}'$

If the summary Abox $\mathcal{A}'$ obtained by applying the mapping function $\mathbf{f}$ to $\mathcal{A}$ is consistent w.r.t. a given Tbox $\mathcal{T}$ and a Rbox $\mathcal{R}$, then $\mathcal{A}$ is consistent w.r.t. $\mathcal{T}$ and $\mathcal{R}$. However, the converse does not hold. In the case of an inconsistent summary, we use a process of iterative refinement to make the summary more precise, to the point where we can conclude that an inconsistent summary $\mathcal{A}'$ reflects a real inconsistency in the actual Abox $\mathcal{A}$. Refinement is a process by which only the part of the summary that gives rise to the inconsistency is made more precise, while preserving the summary Abox properties (1)-(3). To pinpoint the portion of the summary that gives rise to the inconsistency, we focus on the *justification* for the inconsistency, where a justification is a minimal set of assertions which, when taken together, imply a logical contradiction.

We define refinement for a summary individual s in a justification $\mathcal{J}$ as a partition where individuals mapped to s are partitioned based on which edges in $\mathcal{J}$ each individual actually has. More specifically:

$$key(a, \mathcal{J}) \equiv \left\{ R(t,s) \;\middle|\; \begin{array}{l} \mathbf{f}(a) = s \wedge \\ R(t,s) \in \mathcal{J} \wedge \\ \exists b \text{ in } \mathcal{A} \text{ s.t.} \\ R(b,a) \in \mathcal{A} \wedge \\ \mathbf{f}(b) = t \end{array} \right\} \cup \left\{ R(s,t) \;\middle|\; \begin{array}{l} \mathbf{f}(a) = s \wedge \\ R(s,t) \in \mathcal{J} \wedge \\ \exists b \text{ in } \mathcal{A} \text{ s.t.} \\ R(a,b) \in \mathcal{A} \wedge \\ \mathbf{f}(b) = t \end{array} \right\}$$

[4] We assume without loss of generality that $\mathcal{A}$ does not contain an assertion of the form $a \doteq b$

Since an individual may be mapped to a summary individual that is in multiple overlapping justifications, we define:

$$key^*(a) = \bigcup_{\{\mathcal{J} | a \in \mathcal{J}\}} key(a, \mathcal{J})$$

In a *refinement step* that refines s in $\mathcal{A}'$, new individuals $s_1 ... s_k$ replace s in $\mathcal{A}'$, where there are k unique key sets $key^*(a)$, for all a in $\mathcal{A}$ such that $\mathbf{f}(\mathrm{a}) = \mathrm{s}$. Individuals a and b in $\mathcal{A}$ mapped to s in $\mathcal{A}'$ are partitioned correspondingly, that is, $\mathbf{f}(\mathrm{a}) = \mathbf{f}(\mathrm{b})$ after the refinement step iff $key^*(a) = key^*(b)$ before the refinement step.

In principle, in the presence of many justifications involving overlapping sets of nodes, the union of the keys could become very large. In practice, we have not observed this across the various knowledge bases we have evaluated, even for ones that do contain overlapping justifications.

If all individuals in $\mathcal{A}$ mapped to a summary individual s have the same key w.r.t. $\mathcal{J}$, then it must be the case that they have all the edges in the justification and hence s is precise w.r.t. $\mathcal{J}$. If a justification is precise, we can conclude that all individuals in $\mathcal{A}$ mapped to the tested individual in the justification are solutions to the query. In the worst case, iterative refinement can expand a summary Abox into the original Abox, but in practice, we conclude on precise justifications with many individuals mapped to each summary node in the justification.

Our implementation of summarization and refinement in a system called SHER is in terms of RDBMS operations to allow the system to scale to large data sets. However, the iterative process of summarization and refinement is expensive, because (a) it requires expensive join operations on all role assertions in the Abox $\mathcal{A}$ to define the $key(a)$, as well as expensive join operations of role assertions with type assertions to rebuild the summary, and (b) it requires several consistency checks to find the many sources of inconsistencies for each summary that gets built. For large knowledge bases with multiple ways in which one can derive a solution to the query, this becomes a serious performance bottleneck.

3 A Sample Knowledge Base

We illustrate our techniques with the sample knowledge base (Tbox $\mathcal{T}$, the Rbox $\mathcal{R}$ and the Abox $\mathcal{A}$) in Figures 1 and 2. This example is a small subset of the UOBM [3] benchmark that we use in our evaluation. To form the summary Abox for Figure 2, the individuals a and b are mapped to a single summary individual w with a concept set of $Woman$, and the individuals f, g and j are mapped to another summary individual p with a concept set of $Person$. The summary Abox is shown in the Figure 3.

Consider the query $WomanWithHobby$, which is defined as $Woman \sqcap \geq 1 likes$. There are three solutions. The individual a is a solution because $loves \sqsubseteq likes$. The individual f is a solution because the course d can be taught by only one $Person$, and so f and b will be identified with each other during reasoning. Finally, g is a solution, since $isStudentOf(g, WomenCollege)$ implies that g is a $Woman$.

$\mathcal{T}$ assertions:

(1) $WomanCollege \sqsubseteq \forall hasStudent.Woman$
(2) $\top \sqsubseteq \leq 1 isTaughtBy$

$\mathcal{R}$ assertions:

(1) $loves \sqsubseteq likes$
(2) $isStudentOf$ is inverse of $hasStudent$
(3) $teacherOf$ is inverse of $isTaughtBy$

Fig. 1. Example $\mathcal{T}$, $\mathcal{R}$

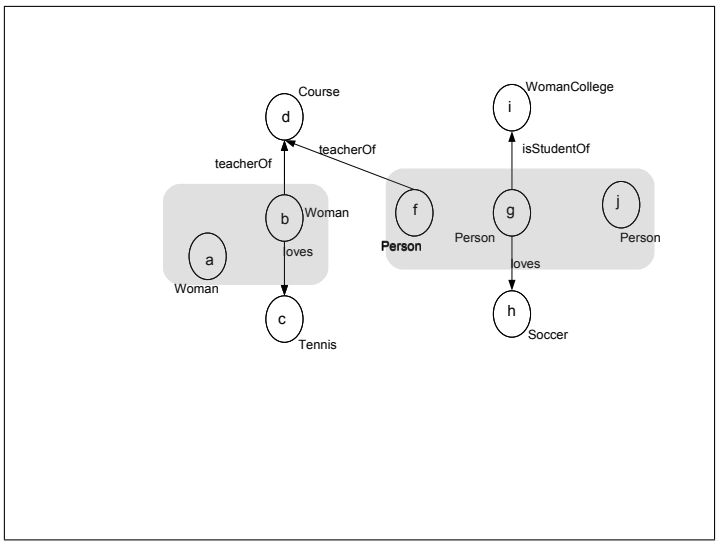

Fig. 2. Example $\mathcal{A}$

Figure 3 shows the entire refinement process for answering this query:

(1) Refine w by splitting it into two nodes w' which has a mapped to it, and w'' which has b mapped to it.
(2) Refine p by splitting it into two nodes p' which has g mapped to it, and p'' which has f and j mapped to it.
(3) Refine p'' further, by splitting it into nodes p_1 which has f mapped to it, and p_2 which has j mapped to it.

We explain these steps in more detail. First, $\neg WomanWithHobby$ is added to a tested summary individual w. The resulting Abox is inconsistent, and a justification $\mathcal{J}$ contains the assertions: $w : Woman$, $loves \sqsubseteq likes$, and $loves(w, c)$. For refinement, we target the summary individuals in $\mathcal{J}$, which are w and c. Refinement makes a justification $\mathcal{J}$ *precise*, that is, it partitions the individuals

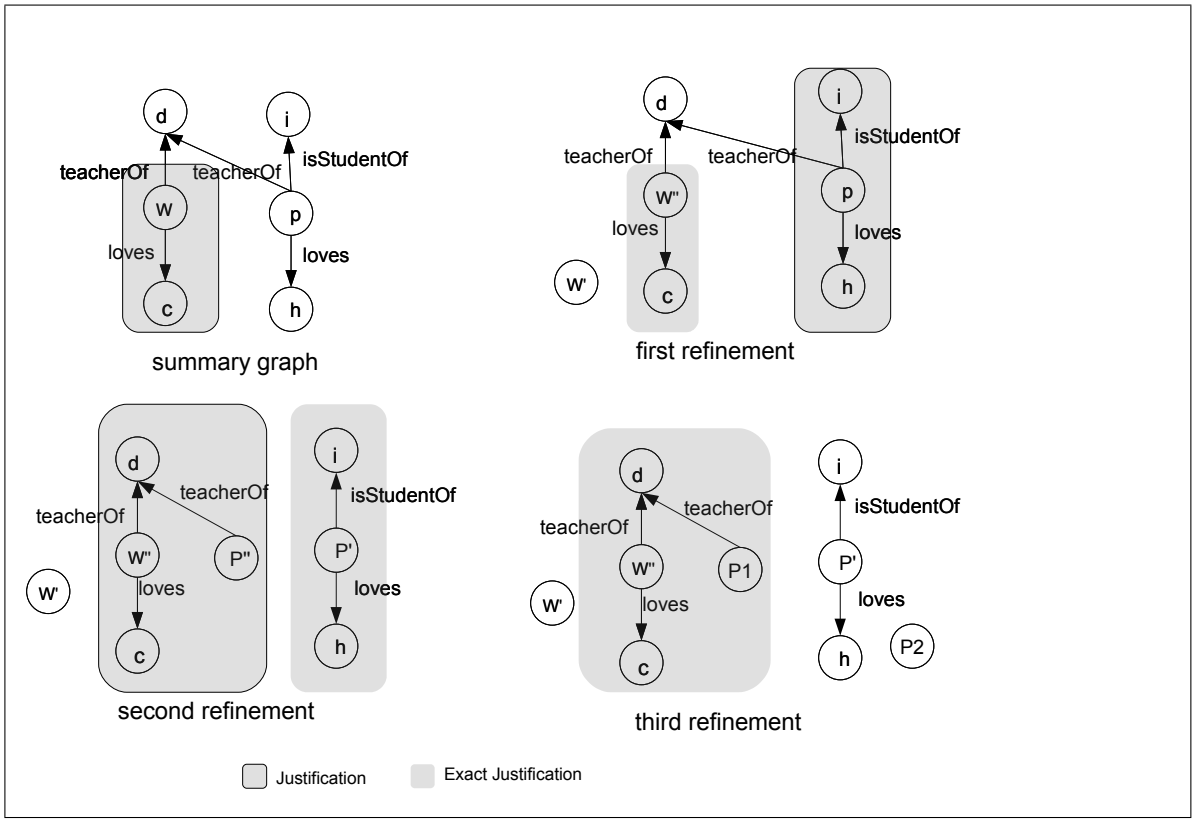

Fig. 3. Refinement Steps for Example

mapped to the summary node w into a new set of summary nodes to reflect the fact that not all individuals in $\mathcal{A}$ mapped to w have the $loves(w, c)$ in $\mathcal{J}$. The summary individual w is therefore split into two new summary nodes, w' that has individuals with no $loves(w, c)$ mapped to it (e.g., a), and w'' that has individuals with $loves(w, c)$ mapped to it (e.g., b). This new refined Abox is still inconsistent, with a new justification $\mathcal{J}$ which contains the individuals w'' and c. Refinement of w'' or c however is no longer possible, because every individual in $\mathcal{A}$ that is mapped to w'' also has the $loves(c, ,)$ and every individual mapped to c has the same edge (here c is the same as the summary node c). At this point, the justification $\mathcal{J}$ is precise, in that it cannot be refined further, and we conclude that all individuals in $\mathcal{A}$ mapped to w'' are solutions to the query.

For the second step, $\neg WomanWithHobby$ is added to a tested summary individual p. The resulting Abox is inconsistent, and this time there is the justification: $isStudentOf(p, i)$, $loves \sqsubseteq likes$, and $loves(p, h)$, combined with the axiom $WomanCollege \sqsubseteq \forall hasStudent.Woman$. The result of the second refinement is shown in Figure 3. After this refinement, the subgraph containing p' is

still inconsistent, and p' is not refinable. Therefore, all individuals in $\mathcal{A}$ mapped to p', namely g, are solutions.

There is one final justification which is refinable: $teacherOf(p'', d)$, $teacherOf(w'', d)$, $w'' : Woman$, $loves \sqsubseteq likes$, $loves(w'', c)$, and $\top \sqsubseteq \leq 1 isTaughtBy$. After the third refinement step, we conclude that f mapped to $P1$ is a solution.

On large knowledge bases, the cost of each additional refinement is significant, so it is critical to reduce the number of refinements. We show in the next sections how our hybrid reasoning approach can reduce the number of refinements for this example.

4 Hybrid Algorithm

The key idea to reducing refinement iterations is to (a) quickly find solutions to the query, (b) refine the summary to isolate these solutions into new summary individuals, and (c) ignore these individuals for the rest of the refinement process. We find solutions quickly by using a sound and incomplete reasoning algorithm which does a form of query expansion described in Section 5. Although we present our implementation of a specific incomplete reasoning algorithm, any sound, incomplete algorithm may be plugged into this technique.

To illustrate the overall idea in terms of our example in Figure 2, we expand our query $WomanWithHobby$ into the query $WomanWithHobby(x) \sqcup (Woman(x) \sqcap likes(x, y)) \sqcup (Woman(x) \sqcap loves(x, y))$. This query matches all pairs of individuals in the Abox bound to both x and y, namely the pair (b, c), and this constitutes our set of known bindings. Our next step is to refine the summary Abox, so that the individuals in the solution, namely b and c, are mapped to distinct new summary individuals. We do this by a refining the summary Abox in a manner similar to that described in Section 2; the only difference is that we now partition the Abox individuals according to whether they were bound to any variable in the query or not, rather than according to key sets. That is, $\mathbf{f}(a) = \mathbf{f}(b)$ after the refinement step iff a and b are mapped to the same summary node before the refinement step and either both or neither a and b are individuals in the set of known bindings. Our algorithm keeps track of the subset of known bindings that actually are answers to the query, which is just b in this case. Next, consistency checking is applied to this refined summary, and any remaining inconsistencies are resolved using the standard iterative refinement and summarization process described in [2].

This approach has a nice property: in cases where the incomplete step actually does find all solutions and the summary itself is consistent, the complete reasoning step may simply be a single consistency check on the refined summary. Since there are no more solutions to be found, the only possible causes of inconsistency are spurious inconsistencies, which are the result of our summarization technique. In practice, we find that the incomplete step captures all solutions on most complex queries. This optimization therefore significantly reduces the number of refinements and makes query answering practical for large Aboxes.

The pseudo-code for overall algorithm is shown in the function MembershipQuery in Figure 4.

Function: MembershipQuery

Input: QueryConcept: $x : C$

```
/* Get incomplete answers from sound but incomplete algorithm, which
   can be translated to SQL                                          */
```
sqlQuery ← BuildQuery(C, x, $\emptyset$);
```
/* Get the bindings for all variables in the expanded query, i.e., x
   (which holds the solutions themselves) and any additional
   variables introduced by existential restrictions                  */
```
result ← execute(sqlQuery);
```
/* Build filtered summary for query answering, which is the basic
   summary Abox A'                                                   */
```
summary ← BuildSummary(C);
```
/* Separate the bindings for x from bindings for any existential
   variables                                                         */
```
sqlsolutions ← getBindings(result, x);
others ← $\bigcup_{v \in vars(result)-x} getBindings(result, v)$;
```
/* Refine summary based on solutions found from SQL                  */
```
sum ← refineSummaryFromSolutions(sum, sqlsolutions ∪ others) ;
```
/* Find all summary nodes in new summary which have sqlSolutions
   mapped to them                                                    */
```
sumSolutions ← getSummaryNodesForSQLSolutions(sum, sqlSolutions);
```
/* complete query answering, using refined summary                   */
```
restsolutions ← solveQuery(sum, allnodes - sumSolutions);
return *sqlsolutions* ∪ *restsolutions*

Fig. 4. Overall optimized complete query algorithm

5 Query Expansion

In our current implementation, our incomplete reasoning algorithm expands a subset of DL constructs, namely, intersections, existentials and unions found in a query concept. (Note that a minimum cardinality of 1 is a special case which is treated as an existential ($\exists R.\top$)). The expansion is based on the fact that for a given query concept C, an instance a can be a member of C if there is directly an assertion $a : C$ in the Abox, if there is an assertion $a : C'$ in the Abox, where C' is a subclass of C, or if a satisfies the definition of C (in an equivalence axiom). Query expansion is performed after the Tbox has been absorbed.

For any given query $a : C$, we generate an abstract representation of the query by recursively traversing the definitions and subclasses of the concept C, and by generating corresponding patterns. The abstract query produced consists

of nested AND and OR expression patterns, where the expression terms are variables, constants, or other patterns. In query expansion, we use the following abstract patterns: AndPattern, OrPattern, TriplePattern, TripleTransitivePattern, TypePattern, MergePattern, and EmptyPattern. The AndPattern and OrPattern express intersections and unions, respectively, and can be arbitrarily nested. The TriplePattern expresses a relation $R(x, y)$, and a TripleTransitivePattern expresses an $R(x, y)$ pattern when $Trans(R)$, i.e. when R is transitive. A TypePattern expresses an $x : y$ assertion as a TriplePattern $rdf : type(x, y)$. A MergePattern has a single parameter, which is any of the other patterns; it matches all individuals matched by the parameter pattern, and additionally any individuals that the analysis has determined must be merged with any individual matched by the parameter pattern. An EmptyPattern is one that returns no solutions.

The abstract query pattern is then translated into SQL, which is fairly straightforward, except in the case of MergePattern and TripleTransitivePattern. These are implemented in the form of datalog rules, and are evaluated by a datalog engine.

For our sample query $x : WomanWithHobby$, we first generate an OrPattern P which signifies all the possible ways in which this query can be expanded. The first disjunct of the OrPattern matches individuals of $WomanWithHobby$ directly, $rdf : type(x, WomanWithHobby)$. In this case, however, the $WomanWithHobby$ type does not appear in the Abox, and so we drop this disjunct. Next we would generate OrPattern disjuncts to match individuals that are in subclasses of $WomanWithHobby$, but in this case there are no subclasses. We find subclasses by calling a standard DL reasoner. The one term that is actually generated comes from recursively processing the definition of $WomanWithHobby$, which is $Woman \sqcap \geq 1 likes$. This generates a set of AndPatterns. The first AndPattern has conjuncts TypePattern $rdf : type(x, Woman)$ and TriplePattern $likes(x, y)$. The second AndPattern has conjuncts TypePattern $rdf : type(x, Woman)$ and TriplePattern $loves(x, y)$, since $likes$ has a subproperty $loves$.

Our query expansion algorithm is provided in pseudo-code in 5. The algorithm uses a set of auxiliary information which are defined below.

conjuncts(c) is the set of conjuncts in an intersection concept

disjuncts(c) is the set of all disjuncts in a union concept

existentialRole(c) is the role in an existential concept

existentialConcept(c) is the concept in an existential concept

rolesInAbox is the set of all roles that actually appear anywhere in the Abox.

typesInAbox is the set of all concepts that appear in the Abox, including any complex types.

subproperties(p) is the set of all subproperties of p, as defined by the Rbox (including p itself)

subclasses(c) is all subclasses of concept p, as defined by the Tbox (including c itself)

```
Function:BuildQuery
Input: concept, var, processedConcepts
Output: AbstractQuery
/* input: 'concept' is concept to expand, 'var' is the current query
   variable, 'processedConcepts' is concepts being done currently.
   */
if concept ∉ processedConcepts then
    switch typeof(concept) do
        /* Also covers minCard concepts with a degree of 1          */
        case existential
            /* 'objectVar' is a new variable to bind the object of
               this existential                                     */
            objectVar ← newAnonVar;
            makeOrPattern(
                pc ← processedConcepts ∪ {concept};
                ec ← existentialConcept(concept);
                for r ∈ rolesInAbox ∩ subproperties(existentialRole(concept)) do
                    if transitive(r) then
                        TripleTransitivePattern(var, r, BuildQuery(ec,
                        objectVar, pc));
                    else
                        TriplePattern(var, r, BuildQuery(ec, objectVar, pc));
                    end
                end
            )
        end
        case union
            makeOrPattern(
                for p ∈ disjuncts(concept) do
                    BuildQuery(p, var, processedConcepts ∪ {concept});
                end
            )
        end
        case intersection
            makeAndPattern(
                for p ∈ conjuncts(concept) do
                    BuildQuery(p, var, processedConcepts ∪ {concept});
                end
            )
        end
        case Primitive
            makeOrPattern(
                for t ∈ typesInAbox ∩ subclasses(concept)) do
                    MergePattern(TypePattern(var, t));
                end
                for t ∈ subclasses(concept)) do
                    if definition(t) then
                        BuildQuery(definition(t), var, processedConcepts ∪ {t});
                    else
                    end
                end
            )
        end
        otherwise
            EmptyPattern;
        end
    end
else
    EmptyPattern;
end
```

Fig. 5. Query expansion algorithm

6 Evaluation

We evaluated our technique on two knowledge bases: the first is a real-world knowledge base, and real queries of clinical data that we had used in previous work[4], and the second is the UOBM benchmark[3]. Our experiments were conducted on a 2-way 2.4GHz AMD Dual Core Opteron system with 16GB of memory running Linux, and we used IBM DB2 V9.1 as our database. Our Java processes were given a maximum heap size of 8GB for clinical data, and 4GB for UOBM.

6.1 Clinical trials dataset

In prior work [4], we reported on the use of expressive reasoning for matching of patient records on clinical trials. The 1 year anonymized patient dataset we used contained electronic medical records from Columbia University for 240,269 patients with 22,561 Tbox subclass assertions, 26 million type assertions, and 33 million role assertions. The 22,561 Tbox subclass assertions are a subset of the a larger Tbox which combines SNOMED with Columbia's local taxonomy called MED for a total of 523,368 concepts. For details of the partitioning algorithm used to define the subset see [4]. Although the expressivity of the SNOMED version we used falls in the EL fragment of DL, the expressivity needed to reason on the knowledge base is $\mathcal{ALCH}$. This is because we have type assertions in the Abox which includes assertions of the type $\forall R.\neg C$, where the concept C is itself defined in terms of a subclass or equivalence axiom. As a concrete example, for a given patient, and a specific radiology episode for the patient, the presence of $ColonNeoplasm$ may be ruled out. $ColonNeoplasm$ has complex definitions in SNOMED (e.g., $ColonNeoplasm \equiv \exists AssociatedMorphology.Neoplasm \sqcap \exists FindingSite.Colon \sqcap ColonDisorder$). We selected the 9 clinical trials we evaluated in our earlier work which are shown Table 1. Table 2 shows the DL version of the queries, in the order shown in Table 1. For query $NCT00001162$, the results shown are for the union of 7 different disorders, only 4 of which are illustrated in Table 2.

Table 3 shows the queries, the number of patients matched to the queries, the time to process the queries in minutes, the time in minutes for our hybrid approach (HTime), the time in minutes for our previous approach (Time), the number of refinements with our hybrid approach (HRefinements) and the number of refinements with our previous approach (Refinements). As can be seen from the table, the hybrid approach reduced the number of refinements to 1 in all cases, which reflects the refinement needed to check that there are no additional solutions after the incomplete algorithm has completed (The one case where 0 refinements occurred was because for that specific query, our expressivity checker decided that no refinement was needed given the specific filtered Abox that was built for the query and the Tbox.) The hybrid approach improved our overall query times from 100.4 mins on average with a standard deviation of 113.7, to 15.6, with a standard deviation of 3.5. This is not surprising, given that the

ClinicalTrials.gov ID	Description
$NCT00084266$	Patients with MRSA
$NCT00288808$	Patients on warfarin
$NCT00393341$	Patients with breast neoplasm
$NCT00419978$	Patients with colon neoplasm
$NCT00304382$	Patients with pneumococcal pneumonia where source specimen is blood or sputum
$NCT00304889$	Patients on metronidazole
$NCT00001162$	Patients with acute amebiasis, giardisis, cyclosporiasis or strongloides...
$NCT00298870$	Patients on steroids or cyclosporine
$NCT00419068$	Patients on corticosteroid or cytotoxic agent

Table 1. Clinical Trial Requirements Evaluated

entire variability in query answering in our previous approach was due to the number of refinements.

6.2 UOBM

We evaluated our approach on the UOBM benchmark, modified to $\mathcal{SHIN}$ expressivity. This was done by adding a new concept to correspond to each of the nominals in the dataset (e.g. SwimmingClass for Swimming), adding a type assertion for each nominal (e.g., $Swimming : SwimmingClass$), and changing any of the references to nominals in the Tbox to point to the class. We are evaluating membership query answering, so we tested one membership query for each concept in the benchmark[5], comparing the hybrid approach with our prior techniques. We report results for UOBM size 100—with roughly 7.8 million type assertions and 22.4 million role assertions—and UOBM size 150—with about 11.7 million type assertions and 33.5 million role assertions. The queries naturally fall into three categories:

empty Concepts that have no instances in the Abox.

simple Concepts that have only simple solutions (i.e. reasoning does not require iterative refinement because the justification viewed as a graph does not have path lengths greater than 1).

complex Concepts that have complex solutions (i.e. reasoning requires iterative refinement because the justification viewed as a graph has path lengths greater than 1).

We expect the hybrid approach to benefit only the third category of queries. One complication is that the summary Abox for the UOBM benchmark has a spurious inconsistency induced by the summarization process, so all membership query answering require 2 passes of refinement in order to make the summary consistent/footnoteThis is a deficiency in our current implementation.

[5] That is, all classes in the original benchmark. The extra classes introduced by our transformation to SHIN are ignored.

DL Query
$\exists associatedObservation.MRSA$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.Warfarin$
$\exists associatedObservation.BreastNeoplasm$
$\exists associatedObservation.ColonNeoplasm$
$\exists associatedObservation.$ $\begin{pmatrix} PneumococcalPneumonia \\ \sqcap \\ \exists hasSpecimenSource.Blood \sqcup Sputum \end{pmatrix}$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.Metronidazole$
$\exists associatedObservation.$ $\begin{pmatrix} acuteamebiasis\sqcup \\ giardisis\sqcup \\ cyclosporiasis\sqcup \\ strongloides\sqcup \\ \ldots \end{pmatrix}$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.cyclosporine \sqcup steroids$
$\exists associatedObservation.$ $\exists roleGroup.$ $\exists administeredSubstance.$ $\exists roleGroup.\exists hasActiveIngredient.corticosteroid \sqcup cytotoxicAgent$

Table 2. DL Queries for Evaluated Clinical Trials

Table 4 shows results for the 3 query categories for UOBM sizes 100 and 150. The first three columns list the UOBM dataset size, the category of query, and how many such queries there are. For both sizes and each query category, we report the average and standard deviation for the query time and the number of passes of refinement. For both datasets, we timed out queries that took longer than 30 minutes to complete; the timeouts occur ed on both the 100 size (1 timeout) and the 150 size (6 timeouts) for the original approach. Hence, those averages and standard deviations are significant underestimates, and so are marked with a $*$ in the table.

As one might expect, there is some overhead for executing the incomplete query, and so the simpler queries actually show some slowdown in the hybrid approach. However, the results do indicate that our hybrid approach does indeed greatly reduce the time for the complex queries, which were the most expensive ones with our previous approach. In fact, for all but one query, the incomplete

Query	Matched Patients	Time (m)	HTime (m)	Refinements	HRefinements
$NCT00084266$	1052	68.9	17.8	6	1
$NCT00288808$	3127	63.8	11.6	5	0
$NCT00393341$	74	26.4	12.1	2	1
$NCT00419978$	164	31.8	12.4	3	1
$NCT00304382$	107	56.4	15.1	8	1
$NCT00304889$	2	61.4	20.7	3	1
$NCT00001162$	1357	370.8	13.5	58	1
$NCT00298870$	5555	145.5	19.3	8	1
$NCT00419068$	4794	78.8	17.5	5	1

Table 3. Patient Matches for Trial DL Queries for 240,269 Patients

			Time (seconds)				Refinement			
			Original		Hybrid		Original		Hybrid	
Size	Category	Count	Average	Stdev	Average	Stdev	Average	Stdev	Average	Stdev
100	empty	11	214	37	214	19	2	0	2	0
100	simple	43	255	83	265	47	2	0	2	0
100	complex	14	891*	386*	377	105	14*	11*	3	.3
150	empty	11	301	35	347	45	2	0	2	0
150	simple	43	340	88	416	85	2	0	2	0
150	complex	14	1368*	508*	647	198	14*	11*	3	.3

Table 4. Results for UOBM Membership Queries for sizes 100 and 150

reasoning algorithm found all the solutions. The one query which was the outlier, `GraduateCourse`, required propagation from a universal restriction for reasoning, which was not accounted for by our incomplete algorithm. In this case, we proceeded to find the answer through our prior complete reasoning algorithm.

7 Related Work and Conclusions

There have been efforts in the semantic web community to define less expressive subsets of OWL-DL for which reasoning is tractable. The EL-family of languages [5] is one such example, for which classification can be done in polynomial time. The fragment $\mathcal{EL}^{++}$ [5] supports nominals, which means that Abox reasoning (i.e. conjunctive query answering) can also be done in polynomial time. However, transforming Abox assertions into nominals-based axioms in the Tbox and doing classification can still be expensive, especially when the Abox contains millions of assertions. To our knowledge, there are no results for this particular scenario; [6] describes performance of the CEL reasoner for classifying the SNOMED Tbox (which is in EL) containing roughly 700K assertions in 29 mins.

Another example is the DL-Lite family [7], for which conjunctive query answering is expressible as a first-order logic formula (and hence an SQL query) over the Abox stored in a relational database, yielding LOGSPACE data-complexity.

[7] describes a query reformulation algorithm, which is similar in spirit to ours, using IS-A, role-typing ($\exists R.A \sqsubseteq C$), and participation ($A \sqsubseteq \exists R.\top$) assertions in the Tbox to expand query concepts.

Finally, the Oracle 11g system supports another subset of OWL known as OWL-Prime and does query answering over the data by precomputing inferences using a forward-chaining rule engine. Using this technique it is able to scale query answering to millions of assertions.

However, a key point is that the optimizations developed in the above approaches, whether it is query-reformulation or the evaluation of a simple rule-set to compute sound, but not necessarily complete, solutions to the query can be plugged into our hybrid system when dealing with the more expressive OWL-DL. When it is known that the optimization is complete based on the underlying logic of the KB[6] and the manner in which it is implemented, fallback to our refinement strategy is not necessary. Otherwise, the refinement process will find any remaining solutions.

References

1. A.Fokoue, A.Kershenbaum, L.Ma, E.Schonberg, K.Srinivas: The summary abox: Cutting ontologies down to size. Proc. of the Int. Semantic Web Conf. (ISWC 2006) (2006) 136–145
2. Dolby, J., A.Fokoue, Kalyanpur, A., A.Kershenbaum, L.Ma, E.Schonberg, K.Srinivas: Scalable semantic retrieval through summarization and refinement. Proc. of the 22nd Conf. on Artificial Intelligence (AAAI 2007) (2007)
3. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y.: Towards a complete owl ontology benchmark. In: Proc. of the third European Semantic Web Conf.(ESWC 2006). (2006) 124–139
4. C.Patel, J.Cimino, J.Dolby, A.Fokoue, A.Kershenbaum, L.Ma, E.Schonberg, K.Srinivas: Matching patient records to clinical trials. Proc. of the Int. Semantic Web Conf. (ISWC 2007) (2007)
5. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence IJCAI-05, Edinburgh, UK, Morgan-Kaufmann Publishers (2005)
6. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL—a polynomial-time reasoner for life science ontologies. In Furbach, U., Shankar, N., eds.: Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06). Volume 4130 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2006) 287–291
7. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: Dl-lite: Tractable description logics for ontologies. Proc. of AAAI (2005)

[6] Checking whether the logic falls in EL or DL-Lite is a matter of syntactic checking of the KB axioms which can be done easily