Optimizations for Answering Conjunctive ABox Queries: First Results

Evren Sirin and Bijan Parsia
Maryland Information and Network Dynamics Lab,
8400 Baltimore Avenue, College Park, MD, 20740 USA
evren@cs.umd.edu, bparsia@isr.umd.edu

1 Introduction

Conjunctive query answering is an important task for many applications on Semantic Web. It is important to efficiently answer queries over knowledge bases with large ABoxes. Although answering conjunctive queries has been studied from a theoretical point of view, until very recently, there was no reasoner supporting such functionality. As a result, there is not enough implementation experience and optimization techniques developed far answering conjunctive queries.

In this paper, we address the problem of answering conjunctive ABox queries efficiently without compromising soundness and completeness. Our focus is on answering queries asked against a very large ABox. We consider queries with only distinguished variables. We focus on the case of distinguished variables because such queries can be answered more efficiently and such queries occur more frequently in realistic scenarios.

We start with the discussion of answering atomic ABox queries and then describe a sound and complete conjunctive query answering algorithm. We then present optimization methods to improve the query evaluation analogous to the optimization methods developed and studied in the context of relational databases. We discuss how the ordering of query evaluation can affect performance, describe some simple yet effective metrics of evaluating the cost of a given ordering and present some heuristics to find (near-)optimal orderings. In the end, we show that the cost model we described provides a sufficiently accurate approximation to the actual query answering time.

2 Answering Atomic ABox Queries

Retrieving Instances The ground query $C(a)$, so-called *instance check*, is answered by adding the negated statement $\neg C(a)$ to the ABox and checking for (in)consistency.

Pseudo model merging technique [2] can be used to test if the pseudo-model of the individual can be merged with pseudo-model of the negated concept $\neg C$ to detect obvious non-instances without doing a consistency test.

The naive way to answer the unground atomic query $C(x)$, so-called *instance retrieval*, is to iterate over all the individuals in the ABox and do a consistency check when the above methods fail to detect an obvious instance or non-instance. Generally, most of the remaining individuals are non-instances and would not cause an inconsistency when the negated statement is added to the KB. *Binary instance retrieval* technique presented in [1] exploits this characteristic and combines many instance checks in one ABox consistency test. If there is no inconsistency all candidates are proven to be non-instances, otherwise the method splits the set of candidates into two sets and continues.

The effectiveness of binary instance retrieval is maximized if obvious instances are found upfront and not put into the candidate list. Model merging technique can be used to detect *obvious instances* by caching dependency set information in the pseudo models. Model merging technique checks for possible interactions between models but finding a clash between models does not mean these models are not mergable because the clash might depend on a non-deterministic choice in tableau completion. There might be other completions of the ABox without the clash. Using the cached dependency set information in pseudo models distinguish a deterministic clash form others and identifies where the models *cannot* be combined in any possible completion meaning that the the individual is necessarily an instance of the concept.

One case where finding obvious instances fails is the case of *defined concepts*. Suppose a concept C is defined as $C \equiv D \sqcap \exists p.A$. Any instance of D that is related to an A instance with a p role is also an instance of C. Unfortunately, above methods would fail to detect such a case. If we are trying to find the instances of a defined concept then breaking up the concept description and applying the above method would help us to find an instance without doing a consistency check. This technique has proved to be very effective in our experiments.

Retrieving Role Fillers In OWL-DL, the role constructors are much less expressive compared to concept constructors. Therefore, verifying $p(\mathsf{a}, \mathsf{b})$ holds only if in the original ABox it is asserted that a and b is related by p or one of its subroles. The interactions between the role hierarchy and number restrictions invalidate this assumption, e.g. a super role assertion combined with cardinality restrictions may cause the relation to hold. Transitive roles complicate the situation even more, now a path between individuals is enough for the relation to hold. Having nominals in the KB completely makes things even more complicated as nominals might relate individuals from disconnected parts of the ABox.

Even if asserted facts in the ABox does not help to find identify all the role assertions between individuals, the completion graph generated for the ABox consistency test can be used to find obvious relations and non-relations. There might be a rela-

tion between disconnected individuals only if such a relation occurs in every model. This suggests by examining the completion graph we can still detect non-relations because if two individuals are disconnected in a completion graph then there is at least one model they are not related and the entailment does not hold. The details of this approach can be found in [4] where we have described this technique in detail for detecting non-subsumptions between concepts.

After finding all obvious relations and non-relations, one might still be left with some possible candidates that might or might not be related. At this point, we can reduce the query $p(x, \mathsf{a})$ (resp. $p(\mathsf{a}, x)$) to an instance retrieval query for concept $\exists p.\{a\}$ (resp. $\exists p^{-}.\{a\}$). If both arguments in the query are unground as in $p(x, y)$, then we first need to generate all candidates for x and then use the above techniques to find corresponding y values.

3 Answering Conjunctive Queries

In this section, we consider answering arbitrary conjunctive queries. W.l.o.g. we assume the query graph is connected. For queries with disconnected components, we can answer each component separately and then take the Cartesian product of generated solutions.

3.1 Query Answering Algorithm

The pseudo-code of the conjunctive query answering algorithm is given in Figure 1. The algorithm simply iterates through all the atoms in the query and either generates bindings for a variable or tests if the previous bindings satisfy the query atom. Generating bindings are done by invoking the instance retrieval function $retrieve$ which in turn might perform several consistency checks as described earlier. Theoretically, testing the satisfaction of a query atom might also require a consistency check. However, as explained in the previous section, most of these tests can be answered without doing a consistency test. Especially, the retrieval operations regarding the role assertions, e.g. $retrieve(\exists p.\{\mathsf{u}\})$, do not typically require any consistency check.

Initially the algorithm is invoked by $AnswerQuery(\mathcal{K}, A, \emptyset, \emptyset)$ where A is an ordering of the atoms in the query. The correctness of this algorithm is quite clear as the satisfaction of every binding is reduced to KB entailment. Thus, this is a sound and complete procedure for answering conjunctive queries.

The efficiency of this algorithm depends very much on the order query atoms are processed. For example, in the query $C(x) \wedge\ p(x, y) \wedge\ D(y)$, suppose C has 100 instances, each instance has one p value and D has 10.000 instances. The ordering $[C(x), p(x, y), D(y)]$ would be much more efficient compared to the ordering $[D(x), p(x, y), C(y)]$. We would do one instance retrieval operation to get 100 instances, find the corresponding p values and test whether these are D instances. The

function $AnswerQuery(\mathcal{K}, A, B, Sol)$
input $\mathcal{K}$ is the input KB, A is a list of query atoms, B is the binding built so far, Sol is the set of all bindings that satisfy the query

if $A = [\,]$ **then return** $Sol \cup \{B\}$
Let a = first(A) and R = rest(A)
Substitute the variables in a based on the bindings in B
if $a = C(\mathsf{v})$ and $\mathcal{K} \models C(\mathsf{v})$ **then return** $AnswerQuery(\mathcal{K}, R, B, Sol)$
if $a = C(x)$ **then**
  for each $v \in$ retrieve(C) **do**
    Let $Sol = AnswerQuery(R, B \cup \{x \leftarrow v\}, Sol)$
if $a = p(\mathsf{v}, \mathsf{u})$ and $\mathcal{K} \models p(\mathsf{v}, \mathsf{u})$ **then return** $AnswerQuery(\mathcal{K}, R, B, Sol)$
if $a = p(x, \mathsf{v})$ (resp. $p(\mathsf{v}, x)$) **then**
  for each $\mathsf{u} \in retrieve(\exists p.\{\mathsf{v}\})$ (resp. $\mathsf{u} \in retrieve(\exists p^{-}.\{\mathsf{v}\})$) **do**
    Let Sol = $AnswerQuery(\mathcal{K}, R, B \cup \{x \leftarrow u\}, Sol)$
if $a = p(x, y)$ **then**
  for each $\mathsf{v} \in retrieve(\exists p.\top)$ **do**
    for each u $retrieve(\exists p.\{\mathsf{v}\})$ **do**
      Let Sol = $AnswerQuery(R, B \cup \{x \leftarrow u\} \cup \{y \leftarrow u\}, Sol)$
return Sol

Figure 1: Pseudo-code for the conjunctive query answering algorithm

second ordering, on the other hand, requires us to iterate over 10.000 individuals and check for a p^{-} value that does not exists for most d instances.

3.2 Cost-based Query Reordering

There are several important challenges to finding an optimal query reordering. In query optimization for relational databases, the main objective is to find an optimal join order and generally the bottleneck is reading data from disk. In a DL reasoner, the most costly operation is consistency checking so we should try to minimize the number of consistency checks performed.

There are two parameters that will help us to estimate the cost of answering a query. First we need to estimate how costly an atomic query is, e.g. for an instance retrieval query, estimate how long it will take to find all the instances, and then estimate the size of results, e.g. how many instances a concept has. These two parameters are interdependent to some degree. For example, if C has 100.000 instances and D has only 10 instances, retrieving C instances can be more costly. However, this is not always true because all the 100.000 individuals might be considered as possible D instances (if the methods described in Section 2 fail) and force us to do expensive consistency tests. For this reason, there is no easy way of estimating these parameters that would work for different ontologies. In the implementation section, we will briefly describe some preliminary methods we devised to compute these parameters.

For now, we will assume that for each atomic query type there are cost functions $\mathcal{C}_{ir}(C)$, $\mathcal{C}_{ic}(C)$, $\mathcal{C}_{rr}(r)$ and $\mathcal{C}_{rc}(r)$ that returns the cost of instance retrieval for concept

C, the cost of a single instance checking for concept C, the cost of role filler retrieval for role r and the cost of verifying a role filler for role r, respectively. Note that, we are assuming the cost of instance checking for a concept is same for all the different individuals in the KB. This assumption may not be very accurate but considering that we will typically deal with large number of individuals, it is not practical to compute estimates for every individual.

In addition, we need to estimate the number of instances of a concept and how many role fillers exist for a given individual and a role. Assuming the size estimates are computed at a preprocessing step, we will use $|C|$ to denote the number of C instances and $|p|$ to denote the total number of tuples in p relation. The average number of p fillers for an individual is denoted by $avg(p)$ and computed as $|p|/|\exists p.\top|$.

Given the parameters for the cost computation and the size estimates, algorithm described in Figure 2 computes an estimate for the cost of query answering for a certain ordering.

Cost estimation is linear in the number of query atoms, provided that size estimates are already computed. However, there are exponentially many orderings to try so an exhaustive search would still be very expensive. Again, as in relational databases, it is possible to use some heuristics to prune the search space. The heuristics we use are: 1) For each atom at position $i > 1$ in the ordered list, there should be at least one atom at position $j < i$ s.t. two atoms share at least one variable. 2) Atoms of the form $p(x, \mathsf{v})$ and $p(\mathsf{v}, x)$ should appear before other atoms involving x. 3) An atom of the form $C(x)$ should come immediately after the first atom that contains x.

First rule is similar to the general query optimization rule that cross products should be avoided. Second rule makes use of the fact that generally an individual is related to limited number of other individuals. And the last rule is to discard the orderings such as $[C(x), p(x, y), q(y, z), D(y)]$. This ordering is not desirable because if $p(x, y)$ finds a binding for y such that $D(y)$ is not satisfied, we would unnecessarily retrieve the q fillers before realizing the failure.

function $EstimateCost(A, B)$
if $A = [\,]$ **then return** 1
Let a = first(A) and R = rest(A)
if $a = C(x)$ and $x \in B$ **then return** $\mathcal{C}_{ic}(C) + EstimateCost(R, B)$
if $a = C(x)$ and $x \notin B$ **then return** $\mathcal{C}_{ir}(C) + |C| * EstimateCost(R, B \cup \{x\})$
if $a = p(x, y)$ and $\{x, y\} \subseteq B$ **then return** $\mathcal{C}_{rc}(p) + EstimateCost(R, B)$
if $a = p(x, y)$ and $\{x, y\} \cap B = \{x\}$ **then return** $\mathcal{C}_{rr}(p) + avg(p) * EstimateCost(R, B \cup \{y\})$
if $a = p(x, y)$ and $\{x, y\} \cap B = \{y\}$ **then return** $\mathcal{C}_{rr}(p) + avg(p^-) * EstimateCost(R, B \cup \{x\})$
if $a = p(x, y)$ and $\{x, y\} \cap B = \emptyset$ **then return**
$\mathcal{C}_{ir}(\exists p.\top) + |p| * \mathcal{C}_{rr}(p) * EstimateCost(R, B \cup \{x, y\})$

Figure 2: Pseudo-code for estimating the cost of an ordering

3.3 Query Simplification

In some cases there might be redundant axioms in a query that can be safely removed from the query without affecting the results. For example, if $C \sqsubseteq D$ then the query $C(x) \wedge D(x)$ is logically equivalent to query $C(x)$. Such redundant atoms do not cause to make additional consistency tests (methods described in Section 2 are quite effective for these cases) but even repeating computationally cheap operations many times causes a noticeable overhead in the end.

The idea behind query simplification is to discover redundant atoms with cheap concept satisfiability tests. But performing too many concept satisfiability tests for simplifications that do not occur frequently in queries is wasteful. For example, simplification based on subsumption of named concepts and roles are nearly never applicable in real world queries or in the benchmarking problems for query answering. We have pinpointed the following two common query simplifications:

- Simplify $C(x) \wedge p(x, y) \wedge D(y)$ to
  - $p(x, y) \wedge D(y)$ if $\exists p.\top \sqsubseteq C$ (*Domain simplification*)
  - $C(x) \wedge p(x, y)$ if $C \sqsubseteq \forall p.D$ (*Range simplification*)

Note that range simplification can also be done even if one of the atoms $C(x)$ or $D(y)$ is missing since we can simply insert $\top(x)$ or $\top(y)$ as an additional atom. In such cases global domain/range restrictions of properties can be directly used and simplification can be done with no subsumption test.

4 Implementation and Experimental Evaluation

The optimization techniques described in this paper have been fully implemented in OWL-DL reasoner Pellet. Right now, size estimation for concepts is done by inspecting the completion graph generated by the initial ABox consistency test. Due to space constraints we will only outline the size estimation algorithm w.r.t concepts but the general idea is same for roles, too.

The size estimation algorithm iterates over a random sample of individuals in the ABox and for each concept tries to determine if this individual is an instance of the given concept without doing a consistency check. The techniques described in Section 2 might return `true`, `false`, or `unknown`. A result of `unknown` means that the individual may or may not be an instance but the reasoner cannot conclude without a consistency test. In such cases, we estimate that with probability κ such individuals would indeed be instances of that concept. Thus, the total estimate for a concept is $|C| = \sigma * (|C_{known}| + \kappa * |C_{unknown}|)$ where σ is the sampling ratio. We also use $|C_{unknown}|$ to have an estimate about $\mathcal{C}_{ir}(C)$ and $\mathcal{C}_{ic}(C)$. If $|C_{unknown}| = 0$ then it means that all the individuals can be retrieved without any consistency test. As $|C_{unknown}|$ increases $\mathcal{C}_{ir}(C)$ would typically increase. Of course there are other factors affecting $\mathcal{C}_{ir}(C)$ but these are not yet considered in our implementation.

In our experiments, we first tested the accuracy of the size estimation. For this purpose, have used the data from Lehigh University Benchmark (LUBM) [5] and ontologies Vicodi and Semintec from [3]. The following tables show the number of individuals in each dataset, the time spent for estimating the size of all the concepts in each dataset, and the mean normalized error over all concepts in the given ontology. For example, an error of 3.6 means that if the actual number of instances for a concept was 200, the algorithm returned 200 ± 7.2. We have changed the sampling percentage from $\%20$ to $\%100$.

		Sampling Percentage				
Dataset	Size	$\%20$	$\%40$	$\%60$	$\%80$	$\%100$
LUBM	55664	3.6	6.7	9.7	11.8	15.0
Semintec	17941	0.9	1.6	2.4	3.2	3.9
Vicodi	16942	1.8	3.4	5.1	6.9	8.6

(a) Time spent in seconds

	Sampling Percentage				
Dataset	$\%20$	$\%40$	$\%60$	$\%80$	$\%100$
LUBM	0.7	0.3	0.6	0.4	0.0
Semintec	6.4	4.4	4.9	3.7	0.0
Vicodi	18.5	11.3	7.6	4.7	0.9

(b) The mean normalized error

As expected, error in size estimation decreases as we inspect more and more individuals. More interestingly, for these ontologies, sizes can be computed with perfect accuracy if all the individuals are inspected. However, computation time also increases. Looking at these results we decided to use a sampling ratio of $\%20$ which yields fairly accurate results with reasonable computation time.

Next, we looked at the effectiveness of the cost model defined in this paper. Query ordering has a significant effect especially when there are many atoms in the query. Therefore, we used three conjunctive queries, Q2 (6 atoms), Q8 (5 atoms), Q9 (6 atoms), from LUBM and Q2 (5 atoms) from Semintec and Q2 (3 atoms) from Vicodi. We generated all the possible query orderings, pruned the orderings based on the aforementioned heuristics and computed the time to answer each query with that ordering. Figure 3 shows the scatter plot of different query orderings where X axis is the estimated cost and Y axis is the actual time it took to generate the answers. The correlation factor for each query is different ranging from low (~ 0.5) to perfect score ($= 1$). More importantly, in each case, the lowest-cost query ordering found by the estimation algorithm is very close to the optimal value.

Next table shows these results in more detail and displays query evaluation time (in milliseconds) for the minimum cost ordering, the minimum time ordering, the

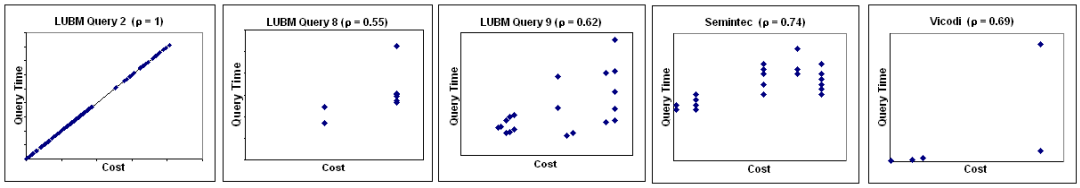

Figure 3: The correlation between the cost estimates and the actual query evaluation time. Each data point represents a different ordering of the corresponding query. Note that, due to simplification and heuristic pruning, number of data points is less than all possible orderings.

maximum time ordering and the median of all orderings (still excluding the heuristically pruned orderings). Although the minimum cost ordering does not always take minimum time, we can still see that the improvement in query evaluation time compared to an arbitrary ordering can be more than one order of magnitude.

	LUBM Q2	LUBM Q8	LUBM Q9	Semintec	Vicodi
Min Cost Ordering	20	320	227	100	81
Min Time Ordering	10	285	164	100	81
Median	1183	341	311	135	255
Max Time Ordering	1233	348	400	140	2123

5 Conclusion and Discussion

In this paper, we have described several different techniques to improve conjunctive query answering for DL reasoners. The query answering algorithm and the optimization techniques we describe do not compromise the soundness and completeness of the reasoner. Furthermore, the additional memory requirement is minimal (linear in the size of TBox but independent of the size of ABox). Our preliminary experimental evaluation shows that the preprocessing time spent for cost estimation is quite reasonable and Pellet can very efficiently answer conjunctive queries over large ABoxes.

There are many open issues left for future work. We are investigating how to estimate the atomic query costs more accurately. Secondly, it is not clear to what extent the effectiveness of our techniques depend on the knowledge base and the query. For our experiments, we picked benchmark datasets that were used in recent publications and considered to be realistic. We are now looking into the evaluation of our techniques on datasets with different characteristics, typically with more complicated TBox components.

References

[1] V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Proc. KR 2004*, 2004.

[2] V. Haarslev, R. Möller, and A.Y. Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In *IJCAR 2001, Italy*, 2001.

[3] B. Motik and U. Sattler. Practical DL reasoning over large ABoxes with KAON2. In *Proc. KR-2006*, 2006.

[4] Evren Sirin, Bernardo Cuenca Grau, and Bijan Parsia. From wine to water: Optimizing description logic reasoning for nominals. In *Proc. KR-2006*, 2006.

[5] Z. Pan Y. Guo and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, 2005.