Towards Incremental Grounding in Tuffy

Wentao Wu, Junming Sui, Ye Liu
University of Wisconsin-Madison

ABSTRACT

Markov Logic Networks (MLN) have become a powerful framework in logical and statistical modeling. However, most of the current MLN implementations are in-memory, and cannot scale up to large data sets. Only recently, Tuffy addresses the scalability issue by using a pure RDB-based implementation.

Inference in Tuffy could be divided into two stages: *grounding* and *search*. The grounding stage substitutes evidence into MLN rules to obtain a set of ground clauses, which is used in the search stage to find the most likely world of the MLN. However, the ground clauses will change if underlying evidence is updated, and redoing grounding from scratch to obtain the new set of ground clauses is costly.

In this paper, we study the problem of incremental grounding in Tuffy. We propose a 2-phase update algorithm, which divides the update procedure into two stages. The first stage tries to find clauses that are unchanged after updating evidence, and can be done completely in memory, while the second stage tries to find remaining new clauses. We prove the correctness of the algorithm, and our experimental evaluation demonstrates its efficiency.

1. INTRODUCTION

Markov Logic Networks (MLNs) [7, 2], as important in a wide variety of modern data management applications such as information extraction [5], coreference resolution [3] and text mining [1], has become a powerful framework in logical and statistical modeling. However, most of the current MLN implementations are in-memory. Thus the size of the data-sets is limited, and the need of scaling up MLNs is of ever growing importance. Only recently, *Tuffy* [6] addresses the scalability issue of inference on MLNs, by using a pure relational database (RDB) based approach.

Inference in Tuffy could be divided into two stages, *grounding* and *search*. In the grounding stage, evidence are pulled out from the database and substituted into each MLN rule to generate a set of *ground clauses*. These ground clauses are stored as materialized views. In the search stage, Tuffy loads the ground clauses into memory, and runs the WalkSAT [4] algorithm to find the most likely world of the MLN.

VLDB '11, August 29- September 3, 2011, Seattle, WA

Now, if we update some evidence in the underlying database, then the ground clauses also change. A straightforward approach is to redo the grounding phase to obtain the new set of ground clauses. However, redoing grounding is costly since we need to examine all possible ground clauses in a very large search space (as reviewed in Section 2). Moreover, if the number of updated evidence is minor, it is likely that most of the clauses before updating the evidence will remain unchanged.

Motivated by these, in this paper, we address the problem of incremental grounding in Tuffy when evidence is updated. We propose a 2-phase update algorithm, and prove its correctness based on a set of properties we identified on the original grounding procedure in Tuffy. We further conduct extensive experiments, which show that our algorithm has considerable performance gain over the baseline strategy by rerunning the grounding procedure.

The rest of the paper is organized as follows. Section 2 reviews basic notions on MLN and Tuffy's grounding stage. Section 3 presents our 2-phase update algorithm. Section 4 gives results from our experimental evaluation. Finally, Section 5 concludes the paper and discusses some future work.

2. PRELIMINARIES

In this section, we briefly review some basic notions related to Markov Logic Network and Tuffy's grounding stage.

2.1 Markov Logic Network

A Markov Logic Network (MLN) is a set of *formulas* $\mathcal{F} = \{F_1, F_2, ..., F_N\}$, with weights $w_1, w_2, ..., w_N$. Each formula could be represented in the form:

$$\phi_1, \phi_2, ..., \phi_m \Rightarrow \phi,$$

or in its *clausal form*:

$$\neg\phi_1 \vee \neg\phi_2 \vee \cdots \vee \neg\phi_m \vee \phi,$$

which are logically equivalent. In the rest of this paper, we use the clausal form representation.

2.2 Tuffy's Grounding Stage

Fix a database schema σ and a domain of constants D. Given an MLN formula $F = l_1 \vee l_2 \vee \cdots \vee l_k$, where each l_i is either a positive or negative *literal*, and let the set of free variables in F be $\bar{x} = \{x_1, ..., x_n\}$, the grounding stage creates a new formula $g_{\bar{d}}$ (called a *ground clause*) for each $\bar{d} \in D^n$, where $g_{\bar{d}}$ denotes the result of substituting each variable x_i of F with d_i. Each constituent in the grounding formula is called a *grounding predicate* or *atom* for short.

P
Alice
Bob

(a) `Smoke(P)`

P1	P2
Alice	Dave
Bob	Dave
Alice	Bob

(b) `Friend(P1, P2)`

Figure 1: Example database

EXAMPLE 1 (RUNNING EXAMPLE). *Suppose we have the following database schema:*

(R1)`Smoke(P)` *(*`Sm(P)` *for short),*
(R2)`Friend(P1, P2)` *(*`Fr(P1,P2)` *for short).*

Figure 1 presents a toy database following the above schema.

The set of MLN formulas is:

(F1)$\neg$`Sm(A)` $\vee$ $\neg$`Fr(A,B)` $\vee$ `Sm(B)`,
(F2)$\neg$`Fr(A,B)` $\vee$ $\neg$`Fr(B,C)` $\vee$ `Fr(A,C)`.

Intuitively, (F1) means "friends of a smoking person also smoke", and (F2) means "friends of friends are friends".

Tuffy uses a pure RDB-based approach to perform grounding, by expressing grounding as a sequence of SQL queries and storing the results as materialized views.

- For each predicate $P(\bar{A})$ in the input MLN, Tuffy creates a relation $R_P(\underline{aid}, \bar{A}, truth)$, where each row a_p represents an atom, aid is a globally unique identifier, $\bar{A}$ is the tuple of arguments of P, and $truth$ is a three-valued attribute that indicates if a_p is true, false, or not specified (in the evidence). These tables are the input to grounding, and Tuffy constructs them using standard bulk-loading techniques.

- The ground clauses are stored in a table $C(\underline{cid}, lits, weight)$ where each row corresponds to a single ground clause. Here, cid is the id of a ground clause, $lits$ is an array that stores the *atom* id of each literal in this clause (and whether or not it is negated), and *weight* is the weight of the clause.

A ground clause is called to be *active* is it can be violated by flipping zero or more active atoms, where an atom is called *active* if its value flips at any point during execution. Tuffy only calculate and maintain the set of active clauses during the grounding stage, which will be used in the later inference stage. In [6], an algorithm based on *active closure* is used to obtain these active clauses. The algorithm works as follows: assume all atoms are inactive and compute active clauses; activate the atoms in the grounding result and recompute active clauses; repeat this process until convergence, resulting in an active closure.

3. A 2-PHASE UPDATE ALGORITHM

We propose a 2-phase update algorithm to achieve incremental grounding in Tuffy. Section 3.1 first introduces necessary notations. Section 3.2 then presents the details of the algorithm, and Section 3.3 shows its correctness. Finally, we discuss some implementation details in Section 3.4.

3.1 Notations

In the following sections, we will use A and C to denote the set of active atoms and active clauses *after grounding*, respectively. We also use $\mathcal{E} = \{E_1, E_2, ..., E_n\}$ to denote the set of original evidence, and use $\mathcal{E}' = \{E_1', E_2', ..., E_n'\}$ to denote the set of updated evidence. Each E_i represents a relation in the database that contains evidence for a predicate in the given MLN.

For convenience of reference, Algorithm 1 outlines the procedure for computing active closure (as discussed at the end of Section 2.2), based on the notations just introduced.

Algorithm 1: Active Closure

Input: $\mathcal{E}$, the set of evidence
Output: A, the set of active atoms; C, the set of active clauses

```
Let E = ⋃_{i=1}^{n} E_i;
A ← ∅, C ← ∅;
repeat
    foreach clause c do
        foreach atom a in c do
            if a ∉ A ∪ E then
                Set a to be false;
            end
        end
        Evaluate c;
        if c is violated then
            Add c to C;
            Add all active atoms in c to A;
        end
    end
until no changes to A and C;
return A and C;
```

3.2 The Algorithm

Our ultimate goal is to determine the new set of active clauses C', with respective to the updated evidence $\mathcal{E}'$. A straightforward approach is to run the closure algorithm from scratch. This may be the only reasonable method in some case (e.g., $\mathcal{E}'$ is completely different from $\mathcal{E}$). However, if we assume $|\mathcal{E}' - \mathcal{E}| \ll |\mathcal{E}|$ (we define $|\mathcal{E}| = \sum_{i=1}^{n} |E^i|$, and $|\mathcal{E}' - \mathcal{E}| = \sum_{i=1}^{n} |E_i' - E_i|$), we may expect that $|C' - C| \ll |C|$. Since we already have C in hand, reinventing the wheel seems inefficient.

We thus seek some potentially more efficient way to compute C'. Since C' can be easily obtained if we know the corresponding set A' of active atoms, can we compute A' *only* based on A and $\mathcal{E}'$? This idea, although quite tempting, has inherent difficulty in practice, since whether an atom is in A' depends on whether it is contained by some $c \in C'$, but we still do not know the set C' yet! In other words, we cannot compute A' independently from C', and vice versa. In fact, we infer that the *interdependence* of A and C is indeed the reason for using an *iterative* procedure to compute the closure in the original grounding phase.

However, although we cannot know the *exact* A' independently from C', computing a subset of A' is possible. Our idea of computing C' then works in two stages:

- In the first stage, we run the closure algorithm confined in the set C, with respect to the updated evidence $\mathcal{E}'$: assume all atoms are inactive and compute active clauses in C; activate the atoms in the result and recompute active clauses in C; repeat this process until convergence. This gives a subset C_r of C that remains active after updating the evidence, which is also a subset of C'. Therefore, the set A_r of active atoms contained in C_r is a subset of A'.

- In the second stage, we run the closure algorithm by initializing A' to A_r, until convergence.

We will prove that the so obtained set of active atoms and clauses are exactly the A' and C', in the next section. Algorithm 2 formally describes this idea in detail.

Algorithm 2: Update Active Clauses

Input: $\mathcal{E}'$, the set of updated evidence; C, the set of old active clauses
Output: A', the set of new active atoms; C', the set of active clauses

```
Let E' = ⋃_{i=1}^{n} E'_i;
A_r ← ∅, C_r ← ∅;
repeat
    foreach clause c ∈ C do
        foreach atom a in c do
            if atom a ∉ A_r ∪ E' then
                Set a to be false;
            end
        end
        Evaluate c;
        if c is violated then
            Add c to C_r;
            Add all active atoms in c to A_r;
        end
    end
until no changes to A_r and C_r;
A' ← A_r, C' ← C_r;
repeat
    foreach clause c do
        foreach atom a in c do
            if a ∉ A' ∪ E' then
                Set a to be false;
            end
        end
        Evaluate c;
        if c is violated then
            Add c to C';
            Add all active atoms in c to A';
        end
    end
until no changes to A' and C';
return A' and C';
```

The difference between the latter idea and the former idea is that now computing A_r can be performed by only considering C and $\mathcal{E}'$, which is independent of C'. By decoupling the procedure in this way, we can obtain the set A_r in a much more efficient way, if C is much smaller than all the clauses that should be considered in grounding. Since we assume the amount of updated evidence is very small, A_r is likely to be close to A', and we can expect only a few number of iterations during the second stage.

3.3 Correctness

In this section, we prove the correctness of our algorithm. We use A'_1 and C'_1 to denote the *true* sets of active atoms and clauses, which can be obtained by running Algorithm 1 with $\mathcal{E}'$ as input. We use A'_2 and C'_2 to denote the sets of active atoms and clauses returned by Algorithm 2. Our goal is to prove that $A'_1 = A'_2$, and $C'_1 = C'_2$.

Two properties of the closure algorithm are the keys to our proof. First, reexamining the steps in the closure algorithm reveals an important fact that, *once a clause is determined to be active, it will remain active in the successive steps*. Formally, we have the following lemma.

LEMMA 1. *Let C_i be the set of active clauses identified in the i-th step of the closure algorithm, then $C_i \subseteq C_{i+1}$.*

PROOF. Let $c \in C_i$. If c is positive, then c can be violated if c *can be* evaluated as false, which means every literal in c can be evaluated as false. We do not care those literals containing evidence, since they will remain the same during the whole procedure. For the rest of the literals, the atoms contained in them must either be active or unknown (i.e., not in evidence, while not be identified as active yet). Note that during the execution of the algorithm, the set of active atoms will keep on growing. So when evaluating c in the $(i+1)$-th step, the only difference compared with the i-th step is that some previously unknown atoms now are known as active. But by setting these new active atoms with the same value as they are in the i-th step, c still can be evaluated as false and hence can be violated. Therefore, $c \in C_{i+1}$. The case that c is negative could be argued in the same way. □

The second property says that, *if we start from a larger set of active atoms, we can obtain a larger set of active clauses, and hence a larger set of active atoms.* We establish this fact in Lemma 2. Here, we use f to denote a unit step in the active closure algorithm (i.e., first obtain the active clauses with respect to the current set of active atoms, and then obtain a larger set of active atoms extracted from the active clauses), and $f(A)$ be the resulting (larger) set of active atoms by applying f on A.

LEMMA 2. *Suppose A_1 and A_2 are two sets of active atoms such that $A_1 \subseteq A_2$. Then $f(A_1) \subseteq f(A_2)$.*

PROOF. Let $C(A_1)$ and $C(A_2)$ be the set of active clauses obtained based on A_1 and A_2, respectively. Then we must have $C(A_1) \subseteq C(A_2)$. To see this, consider any active clause $c \in C(A_1)$. Along the analysis in the proof of Lemma 1, we know that c will remain active if we enlarge the set of active atoms. Since $A_1 \subseteq A_2$, we have $c \in C(A_2)$, and hence $C(A_1) \subseteq C(A_2)$. It is then straightforward to conclude that $f(A_1) \subseteq f(A_2)$, since $f(A)$ is obtained by extracting active atoms from $C(A)$ and any atom can only change its state from unknown to active. □

It's now easy to prove the correctness of our update algorithm (Theorem 1).

THEOREM 1. *Let A'_1, C'_1 and A'_2, C'_2 be the sets of active atoms and clauses returned by Algorithm 1 and 2, respectively. Then $A'_1 = A'_2$, and $C'_1 = C'_2$.*

PROOF. Clearly, we only need to prove that $A'_1 = A'_2$, and $C'_1 = C'_2$ is then naturally resulted. We use $f^*(A)$ to denote the closure of active atoms by applying the algorithm on top of A, namely, $f^*(A) = f...f(f(A))$. Then $A'_1 = f^*(\emptyset)$, and $A'_2 = f^*(A_r)$.

In the following proof, we use SC_1^i and SC_2^i (SA_1^i and SA_2^i) to denote the set of active clauses (atoms) obtained in the i-th iteration of Algorithm 2 and the *first stage* of Algorithm 1, respectively.

Consider the first stage of Algorithm 2. During the first iteration, we obtain a subset SC_2^1 of C containing active clauses by setting all atoms other than evidence in $\mathcal{E}'$ to be false. The first iteration of Algorithm 1 does exactly the same job, and results in a set SC_1^1 of active clauses. We hence have $SC_2^1 \subseteq SC_1^1$. According to

Lemma 1, $SC_1^1 \subseteq C_1'$. Therefore, $SC_2^1 \subseteq C_1'$. The resulting sets of active atoms SA_2^1 and SA_1^1 also satisfy $SA_2^1 \subseteq SA_1^1$. In the second iteration of the first stage, the obtained subset SC_2^2 of C is still a subset of SC_1^2, and hence a subset of C_1'. This is because any $c \in SC_2^2$ must also appear in SC_1^2, since Algorithm 1 uses a larger set SA_1^1 (than SA_2^1) of active atoms to evaluate c. Continuing this argument, we know that each iteration of the first stage of Algorithm 2 will produce a subset of C_1', including the final C_r. Thus, $A_r \subseteq A_1'$.

Now consider the second stage of Algorithm 2. Since $\emptyset \subseteq A_r$, according to Lemma 2, we have $f(\emptyset) \subseteq f(A_r)$. Continuing applying Lemma 2, we can obtain $f^*(\emptyset) \subseteq f^*(A_r)$, namely $A_1' \subseteq A_2'$. On the other hand, it must hold that $A_2' \subseteq A_1'$, since A_1' is the *true* closure of active atoms. This completes the proof of $A_1' = A_2'$. □

3.4 Implementation

We discuss some implementation details in this section.

3.4.1 Stage 1

Currently we assume that the set of active clauses after grounding can resident within memory, and Stage 1 uses a very efficient in-memory implementation. This is not always the case in practice. Tuffy [6] further proposes partitioning strategies that try to decompose the MRF after grounding the MLN into disjoint components so that the inference can proceed in a component-by-component style, which only requires sufficient memory for a single component instead of the whole MRF network. We leave the problem of partitioning in Stage 1 as one of our future work.

3.4.2 Stage 2

Same as the grounding procedure in Tuffy, Stage 2 is implemented with a sequence of SQL queries. The goal of Stage 2 is to find those active clauses that are previously *inactive* before updating the evidence. However, according to Theorem 1, the set C' of active clauses from Stage 2 actually contains all the active clauses in C_r that we have found during Stage 1, namely, $C_r \subseteq C'$. Therefore, in the case that C_r is large, it wastes effort in Stage 2 on finding duplicated active clauses. What's more, since the active clauses found in Stage 2 need to be loaded into memory for later inference, it also wastes I/O's in loading this duplicates.

We use the following lemma to further reduce the number of duplicates found in the second stage:

LEMMA 3. *A previously* inactive *clause $c = l_1 \vee \cdots \vee l_k$ will remain* inactive *if:*

For each literal l_i in c, the atom a_i contained by l_i ($l_i = a_i$ or $l_i = \neg a_i$) is in nether ΔE nor ΔA, where ΔE and ΔA is the set of changed *evidence and active atoms, respectively.*

PROOF. Suppose c is positive. The fact c is inactive means that c is evaluated to be true before updating the evidence. Therefore at least one literal l_i in c is true.

(1) If $l_i = a_i$, then a_i must be some positive evidence. Since $a_i \notin \Delta E$, l_i remains to be true and c remains to be inactive.

(2) If $l_i = \neg a_i$, then a_i can be either some negative evidence or some atom not known as active. The case that a_i is some negative evidence can be analyzed in the same way as in (1). On the other hand, if a_i is some atom not known as active before, a_i will remain unknown to be active after updating the evidence, since $a_i \notin \Delta A$. Therefore l_i will still remain to be true and c remains to be inactive.

This completes the proof of the lemma. □

Lemma 3 claims that, a previously inactive clause can turn from inactive to be active only if some of its literals contain elements in $\Delta E \cup \Delta A$. By pushing this condition into the `WHERE` statement in our SQL implementation, we can rule out a lot of inactive clauses that should be considered before. Meanwhile, this condition also has the side effect on pruning active clauses generated in Phase 2. However, although we can still reserve all the inactive clauses that become active after updating the evidence, we need to reexamine the previously active clauses in memory after we obtain the new set of active atoms to ensure that we will not miss any active clause, since the active clauses generated in Phase 2 is now no longer guaranteed to cover all the active clauses after updating the evidence.

The correctness of the algorithm is not affected by applying the above strategy. We admit here that the effect of this strategy depends on how many duplicated active clauses can be pruned, and we leave a deeper study on this problem as our another future work.

3.4.3 Discussion on Performance Improvement

Compared with the straightforward method which simply reruns the grounding procedure from scratch, we improve the performance in the following three aspects:

1. Stage 1 can be completely performed in memory and should contain most of the active clauses after updating the evidence if the update is minor.

2. Stage 2 is started from a larger set of active atoms obtained in Phase 1, hence fewer number of iterations could be expected when running the iterative procedure.

3. The pruning strategy based on Lemma 3 shrinks the search space that should be considered in Phase 2, and also eliminates duplicated active clauses generated by Phase 2, which saves I/O's on loading these clauses from disk into memory.

4. EXPERIMENTAL EVALUATION

In this section, we report the results of our empirical study. We first verify the number of changed active clauses when the evidence is updated. The result matches our intuition that most active clauses will remain unchanged if the update to evidence is minor. We then evaluate our method on both time and space efficiency. Our evaluation shows that, the execution time of our method is about 65% less than the baseline method that reruns the grounding procedure from scratch, while both methods consume comparable amount of memory.

4.1 Experiment Settings

We implement our 2-phase update algorithm on top of the latest Tuffy system (version 0.2) [1]. All the experiments are performed on a desktop PC configured with Intel Xeon E5520 (2.26 GHz) CPU and 12GB of memory, running Redhat Linux, Version 6. Since Tuffy is implemented with Java, we allocate 3GB of memory to the Java Virtual Machine.

We use three datasets in our experiments, all of which are publicly available from the ALCHEMY website[2]. **CSE** consists of data from the administrative database of Computer Science Department at University of Washington, and the query is about the *student-adviser* relationships. **Class** is about publication, and the task is to predict the category of paper, given the author and citation information. **IE** is a collection of different citations to computer science

[1] http://research.cs.wisc.edu/hazy/tuffy/download/

[2] http://alchemy.cs.washington.edu/

Table 1: The number of active clauses that are changed by updating different fraction of evidence

DATASET	%EVIDENCE CHANGED	TOTAL #ACS	# ACS CHANGED	%ACS CHANGED
CSE	1%	363063	11	0.00%
	10%	359389	425	0.12%
	20%	355671	1400	0.39%
CLASS	1%	243799	393	0.16%
	10%	244406	3675	1.50%
	20%	245935	8156	3.32%
IE	1%	8882	0	0.00%
	10%	8190	0	0.00%
	20%	7493	0	0.00%

research papers. Given the fields of author, venue, title and year, the goal is to extract the citations that refer to the same paper.

4.2 Change of Active Clauses

One important motivation for our 2-phase update algorithm is that most of the active clauses may remain unchanged if only minor evidence is updated. Phase 1 hence can find most of the active clauses, and Phase 2 can only focus on the small number of *missing* clauses so that efficient pruning strategy can be applied.

We verify this intuition by studying the change of active clauses on the datasets we used. Since the number of predicates, the number of evidence, and the number of clauses are different for different datasets, the absolute number of changed evidence is not a reasonable choice for measuring the changes. Instead, we consider the fraction of evidence that are changed in all available evidence. (We do not change evidence from predicates that have closed world assumption (CWA), i.e., all ground atoms of this predicate not listed in the evidence are false.)

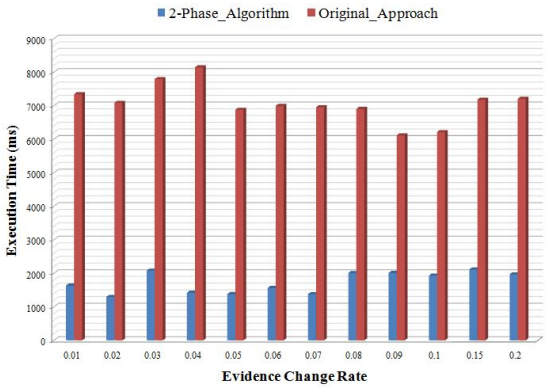
We vary the fraction of changed evidence from 1% to 20%. For each fixed fraction and each predicate table, we uniformly pick up evidence tuples to update, with equal likelihood on `INSERT` and `DELETE` operation. (Since any `UPDATE` operation on an evidence can be accomplished by performing a `DELETE` followed by an `INSERT`, we do not need to consider `UPDATE` operations here.) We then compute the number of active clauses based on the updated evidence, by rerunning the grounding procedure of Tuffy on the updated evidence. Table 1 gives the results.

As we can see from Table 1, when we change 1% of evidence in CSE, only 11 active clauses are changed. Therefore, running grounding procedure from scratch means that we have to generate the other 363K active clauses again from disk, which are actually already in memory. Even we change 20% of the evidence, the number of changed active clauses is still only 1.5K, much smaller than the total number of active clauses. We can also observe similar trend on the Class dataset. Meanwhile, changing the evidence in the IE dataset doesn't change the active clauses, since all evidence provided are from predicates holding CWA.

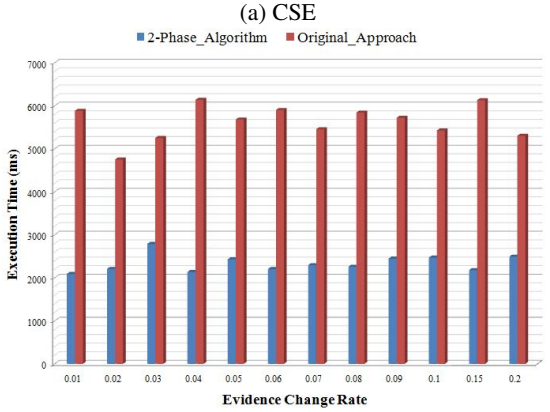
4.3 Time and Space Efficiency

We measure the time and space efficiency of our 2-phase update algorithm, by comparing it with the baseline method which reruns the grounding procedure from scratch. For time efficiency, we first measure the end-to-end execution time for each method, from the time when evidence are updated to the time when all new active clauses are loaded into memory for later inference. We then report the time spent on each phase of our method as well. For space efficiency, we measure the peak memory used by each method.

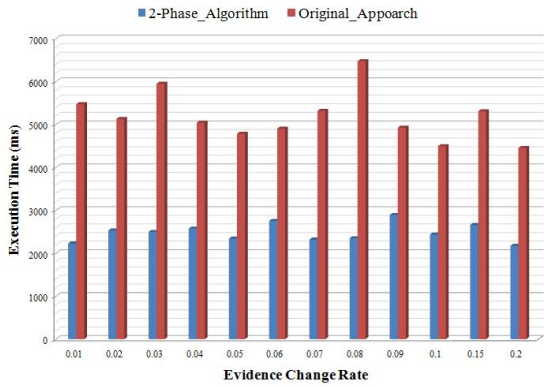
Figure 2 shows the results on end-to-end time elapsed when varying the fraction of updated evidence from 1% to 20%. Our 2-phase algorithm consistently outperforms the baseline method, across the datasets we used. On the CSE dataset, our algorithm is about three to four times faster than the baseline algorithm. On the Class and IE dataset, 2-phase algorithm also saves more than 50% execution time. Considering the number of changed active clauses shown in Table 1, most of the active clauses remain unchanged and hence could be found in memory during Phase 1, the speed-up of our method is expected.



(a) CSE



(b) Class

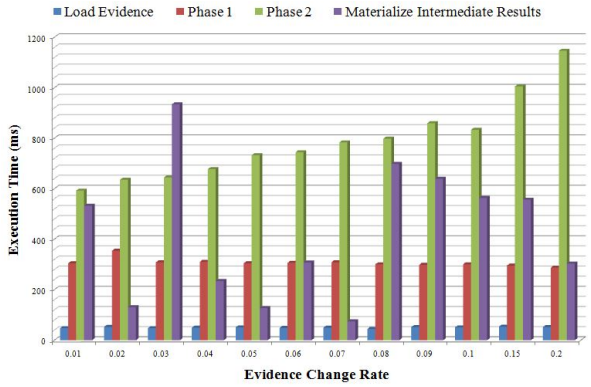


(c) IE

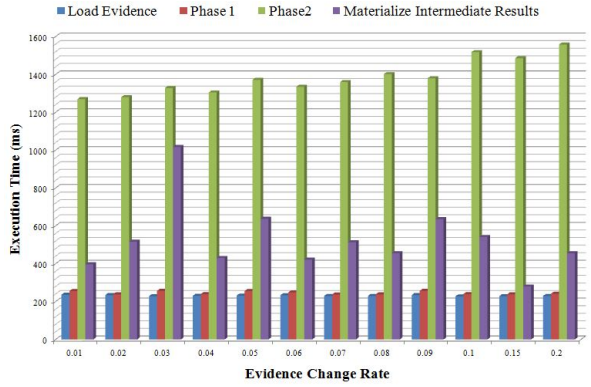
Figure 2: Execution time of 2-phase algorithm and baseline algortihm.

We also investigate the time spent on each individual step of our 2-phase update algorithm. As can be seen from Figure 3, it's expected that Phase 2 costs much more time than Phase 1, since Phase 1 can be done completely within memory. Meanwhile, Figure 3

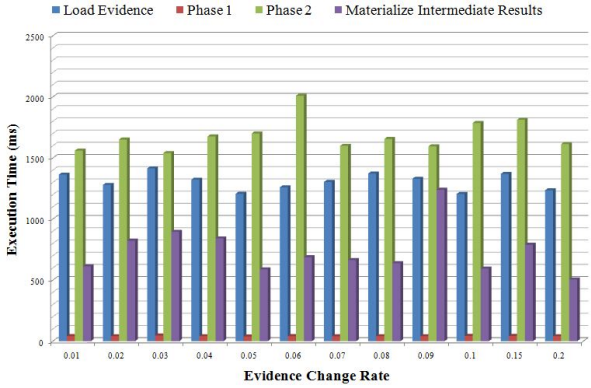
also reveals the fact that considerable amount of time is actually spent on loading evidence into memory (since we need to evaluate the clauses that are violated) and materializing intermediate results onto disk (e.g., the set of active atoms found in Phase 1, which will be used as the seed set of active atoms in Phase 2), due to our SQL implementation of Phase 2. These additional overhead may be avoided if we use more efficient implementations (e.g., by using a more compact representation of the clauses so that the evidence information can be encoded in).



(a) CSE



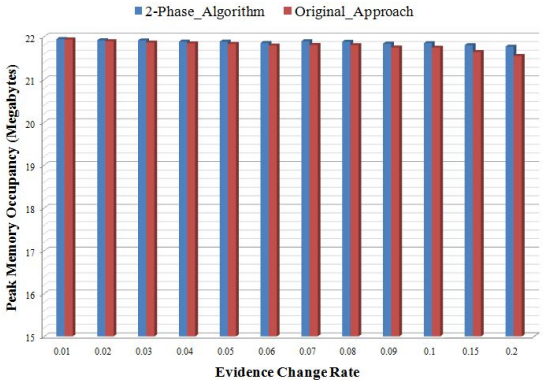
(b) Class



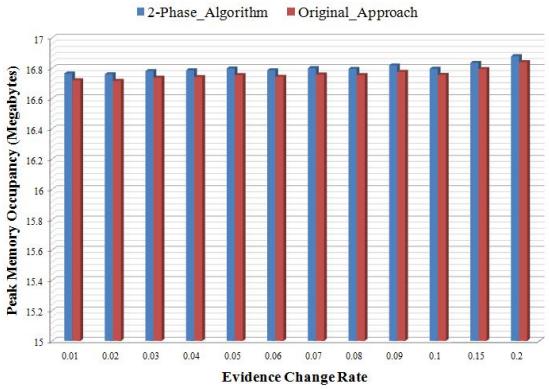
(c) IE

Figure 3: Execution time on each step of 2-phase algorithm.

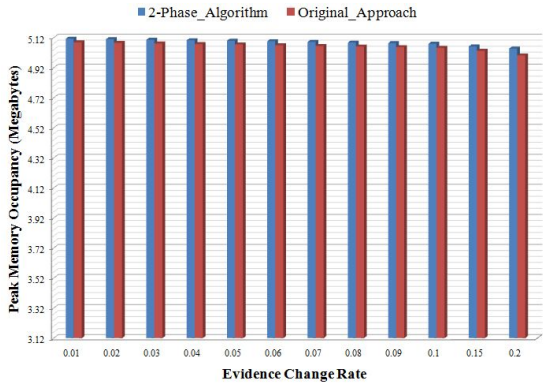
We further compare the peak memory used during the execution of our method and the baseline method. As shown in Figure 4, although our 2-phase algorithm significantly accelerates the updating process, it does not significantly increase the amount of memory used. The reason is that most of the used memory is actually occupied by the set of active clauses, since they need to be load into memory at the end of each method, for later inference. This memory cost is the same for both method. Additional memory cost in both methods are significantly smaller.



(a) CSE



(b) Class



(c) IE

Figure 4: Peak memory occupancy of 2-phase algorithm and baseline algortihm.

5. CONCLUSION AND FUTURE WORK

In this paper, we focuse on the problem of incremental grounding in Tuffy. We propose a 2-phase update algorithm, formally prove its correctness, and conduct extensive experiments that demonstrate its efficiency.

As future work, we will explore the following three directions:

1. As mentioned in Section 3.4.1, partitioning strategy is used in Tuffy to further improve performance. By decomposing the MRF into disjoint components, inference can proceed in a component-by-component manner, which saves both memory and time. Clearly, updating the evidence will also change the components of the MRF, and we are interested in incrementally recompute these components.

2. As mentioned in Section 3.4.2, we will further study the effect of our pruning strategy on different updates made to the MRF.

3. We have verified our intuition that minor update to the evidence will only slightly change the MRF. It could then also be expected that the inference results may also not be changed to much, by using the new MRF. Since statistical inference only gives estimated results, they do not need to be perfectly accurate. As a result, one interesting research issue is to investigate which changes on the MRF will lead to significant impact on the inference results. It can be imagined that most updates on the MRF will not affect the inference results to a large extent, and hence can be ignored if we only seek some good approximation (e.g., by requiring the results within a specified threshold of error). In the best situation, the set active clauses in memory is already sufficient to give such an approximation, and Phase 2 then can be completely discarded.

6. REFERENCES

[1] P. Domingos. Toward knowledge-rich data mining. *Data Mining and Knowledge Discovery*, 15(1):21–28, 2007.

[2] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool Publishers, 2009.

[3] S. Huang, Y. Zhang, J. Zhou, and J. Chen. Coreference resolution using Markov Logic Networks. In *Proceedings of the 10th International Conference on Intelligent Text Processing and Computational Linguistics*, Mexico city, Mexico, 2009.

[4] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In *The Satisfiability Problem: Theory and Applications*, pages 573–586. American Mathematical Society, 1996.

[5] S. Kok and W.-t. Yih. Extracting product information from email receipts using markov logic. In *Proceedings of the Sixth Conference on Email and Anti-Spam*, pages 573–586, Mountain View, California, 2009.

[6] F. Niu, C. Re, A. Doan, and J. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *PVLDB*, 2011.

[7] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.