Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS

Feng Niu Christopher Ré AnHai Doan Jude Shavlik
University of Wisconsin-Madison
{leonn,chrisre,anhai,shavlik}@cs.wisc.edu

ABSTRACT

Over the past few years, Markov Logic Networks (MLNs) have emerged as a powerful AI framework that combines statistical and logical reasoning. It has been applied to a wide range of data management problems, such as information extraction, ontology matching, and text mining, and has become a core technology underlying several major AI projects. Because of its growing popularity, MLNs are part of several research programs around the world. None of these implementations, however, scale to large MLN data sets. This lack of scalability is now a key bottleneck that prevents the widespread application of MLNs to real-world data management problems.

In this paper we consider how to leverage RDBMSes to develop a solution to this problem. We consider ALCHEMY, the state-of-the-art MLN implementation currently in wide use. We first develop BTUFFY, a system that implements Alchemy in an RDBMS. We show that BTUFFY already scales to much larger datasets than Alchemy, but suffers from a sequential processing problem (inherent in Alchemy). We then propose CTUFFY that makes better use of the RDBMS's set-at-a-time processing ability. We show that this produces dramatic benefits: on all four benchmarks CTUFFY dominates both Alchemy and BTUFFY. Moreover, on the complex entity resolution benchmark CTUFFY finds a solution in minutes, while Alchemy spends hours unsuccessfully. We summarize the lessons we learnt, on how we can design AI algorithms to take advantage of RDBMSes, and extend RDBMSes to work better for AI algorithms.

1. INTRODUCTION

Over the past few years, Markov Logic Networks (MLNs) have emerged as a powerful and popular framework that combines logical and probabilistic reasoning. It has been successfully applied to a wide variety of data management problems, including information extraction, entity resolution, and text mining. In fact, it has become a core technology underlying several large projects, such as the DARPA CALO and Machine Reading five-year programs. The framework has also been implemented at many academic and industrial places, including Stanford, SRI, Washington, Wisconsin, and the University of Massachusetts at Amherst.

Unfortunately, none of the current MLN implementations scales beyond relatively small data sets (and even on many of these, they routinely take hours to run). The first obvious reason is that these are *in-memory* implementations. As such, they thrash badly when generating and manipulating large intermediate data structures that overflow the main memory, which they very often do.

Consequently, there is an emerging effort across several research groups to try to scale up MLNs. In this paper, we explore the first obvious candidate, an RDBMS, to manage the memory overflow problem. In particular, DARPA, the research arm of the US Department of Defense, is currently running a large five-year program called "Machine Reading". The goal is to build software that can "read" the Web, i.e., extract and integrate structured data (e.g., entities, relationships) from Web data, then use this structured data to answer user queries. Within this program, the SRI team, a large research group spanning six universities and two industrial labs (and across the AI and database communities), is considering how to apply MLNs to the machine reading problem. To do so, it is critical that MLNs scale to large data sets, and we have been put in charge of investigating this problem, partly because we are also interested in being able to run MLNs to extract and integrate data within our own Cimple/DBLife project.

This paper describes the result of our current investigation. We seek to answer the following questions: Can an RDBMS help scale up MLNs? If yes, by how much, and what are the key reasons (beside the ability to manage data that is larger than the main memory)? Given these reasons, in general how can we design AI algorithms to take advantage of RDBMSes? And how can we extend RDBMSes to work better for AI algorithms? As ever more powerful AI reasoning frameworks are being developed, answers to the above questions can help scale not just MLNs, but those frameworks as well. Further, over the past decade the database community has made a considerable effort to extend RDBMSes to deal with imprecise data, by pushing statistical models into RDBMSes [1–4, 8, 13, 25, 33]. Our work on pushing MLNs into RDBMSes falls into, and can help advance, this emerging direction.

We began our investigation by designing BTUFFY, a relatively straightforward implementation of MLNs in an RDBMS. We chose PostgreSQL because we want to eventually make

VLDB '10, September 13-17, 2010, Singapore

the code open source. Briefly, we store most of the data (original, intermediate, and final) in relational tables, and execute most of the reasoning using SQL commands and queries (§3). Reasoning with MLNs can be classified as either learning or inference [21]. We focus on inference because that tends to be performed multiple times online and so must be fast.

Next, we performed an extensive analysis of BTUFFY. As a comparison point, we select ALCHEMY, the currently most popular and state-of-the-art MLN implementation [5, 21]. On a diverse set of tasks, we found that BTUFFY significantly outperforms ALCHEMY. As expected, one of the reasons is the ability to manage external data: whenever intermediate data is larger than main memory, ALCHEMY thrashed badly whereas BTUFFY did not. But there is more. It turned out that ALCHEMY performs many "relational operation"-like steps (e.g., select, project, join) and that these operations are often a bottleneck in MLN inference: on a classification benchmark over 96% of ALCHEMY's execution time is spent performing relational-like operations. To execute these operations, ALCHEMY uses fixed algorithms (e.g., always using nested-loop join). In contrast, by exploiting the query optimization strength of the RDBMS, BTUFFY finds a good way to execute these steps, thereby significantly speeding up MLN inference.

On the other hand, we found that BTUFFY executed too many SQL queries (as many as millions of queries on certain tasks), and this caused considerable overhead. This happens because when ALCHEMY searches within a large space for the "world" with the highest probability (using WALKSAT [10]), it performs an *inherently sequential* search. It flips the value of a variable, does "book keeping" for a fair amount of time, determines based on that which variable to flip next, flips that variable, does book keeping again, and so on. Consequently, the natural solution of implementing each step of flipping and book keeping with a SQL query ends up generating too many SQL queries, which must be executed sequentially.

To address this problem, we build on the MAXSAT literature to design a new search algorithm where, conceptually, multiple steps of flipping and book keeping can be executed in parallel, as a single group. We then implement each group with a single SQL query, letting the RDBMS decide how best to execute the steps within the group. As such, this algorithm exploits the set-at-a-time-processing strength of RDBMSes. Further, it runs drastically fewer SQL queries, thereby significantly reducing the overhead compared to BTUFFY. We refer to the RDBMS-based MLN solution using this new search algorithm as CTUFFY.

On a diverse set of MLN tasks obtained from the Web (including all applicable MLN tasks from the ALCHEMY repository), we found that CTUFFY dramatically outperforms both BTUFFY and ALCHEMY: on an entity resolution task, CTUFFY finds an optimal cost solution in 11 minutes versus more than 7 hours for ALCHEMY or 2 hours for BTUFFY.

To summarize, we make the following contributions:

- We developed a solution to push MLNs into RDBMSes, a problem of growing interest to both the database and AI communities. As far as we know, ours is the first solution to this problem.
- We presented extensive experiments on diverse MLN tasks. The results show that CTUFFY, our RDBMS-based solution, outperforms ALCHEMY, the state-of-the-art solution, by two orders of magnitude on all but the smallest dataset.
- We identified ways that help make AI algorithms more amenable to processing in an RDBMS, such as avoiding inherently sequential processing steps, and ways to extend RDBMSes to better process AI algorithms, such as lowering the overhead to access main-memory-resident data.

Related Work. MLNs are an integral part of state-of-the-art approaches in a variety of applications: *natural language processing* [16, 17, 22], *ontology matching* [35], *information extraction* [15], *entity resolution* [28], *data mining* [30], *computer vision* [32], *network traffic analysis* [36], and *smart automobile design* [31]. And so, there is an application push to support MLNs.

More broadly, pushing other statistical reasoning models inside a database system has been a goal of many projects [4, 8, 9, 20, 33]. Most closely related is the BAYESSTORE project, in which the database essentially stored *Bayes' Nets* [14] and allowed these networks to be retrieved for inference by an external program. In contrast, TUFFY pushes the inference procedures inside the RDBMS. The Monte-Carlo database [8] made sampling a first-class citizen inside an RDBMS. In contrast, in TUFFY our approach can be viewed as pushing classical search inside the database engine. One way to view an MLN is a compact specification of a *factor graphs* [24, 25]; our results indicate that explicitly creating this graph is untenable on all but the smallest problems. Additionally, TUFFY finds the most likely world, while Sen et. al [25] consider the related, but different, problem of joint probabilistic inference.

There has also been an extensive amount of work on *probabilistic databases* [1–3, 19] that deal with simpler probabilistic models. In fact, finding the most likely world is trivial in these models; in contrast, it is highly non-trivial in MLNs (in fact, it is NP-hard.). Moreover, none of these approaches deal with the core technical challenge TUFFY addresses, which is handling search inside a database. Additional related work can be found in §A.

2. PRELIMINARIES

We illustrate a Markov Logic Network program using the example of classifying papers by topic area. We then define the semantics of MLN, and the current state-of-the-art implementation, ALCHEMY.

2.1 The Syntax of MLNs

Fig. 1 shows an example input MLN program for CTUFFY that is used to classify paper references by topic area, such as databases, systems, AI, etc. In this example, we are given a set of relations that capture information about the papers in our dataset: the authors have been extracted and stored in the relation `wrote`(Person,Author), and the citations have been extracted into the relation `refers`(Paper,Paper), e.g. the evidence shows that Joe wrote papers P1 and P2 and the former cited another paper P3. In the relation `cat`, we are given a subset of papers and which category they fall into. The `cat` relation is incomplete: some papers do not

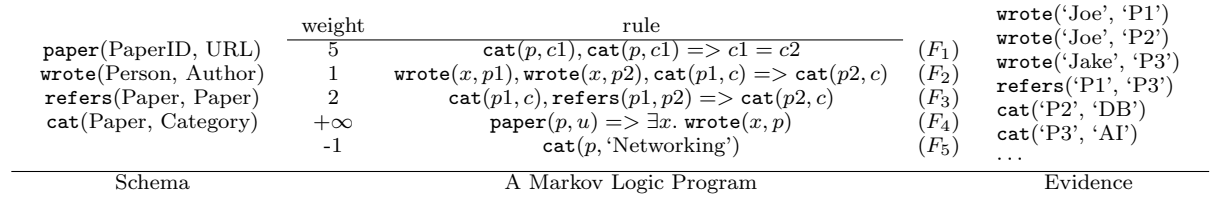

Figure 1: A Sample Markov Logic Program: The goal is to classify papers by category. As evidence we are given author and citation information of all papers, as well as the labels of a subset of the papers; and we want to classify the remaining papers. Any variable not quantified is free.

have a label. We can think of each possible labeling of these papers as an instantiation of the `cat` relation, which can be viewed as a *possible world* [6]. The classification task is to find the most likely labeling of papers by subject area, and hence the most likely possible world.

To tell the system which possible world we should produce, the user provides (in addition to the above data) a set of rules that incorporate their knowledge of the problem. A simple example rule is F_1[1]:

$$\texttt{cat}(p, c1), \texttt{cat}(p, c2) \Rightarrow c1 = c2 \quad (F_1)$$

Intuitively, F_1 says that a paper should be in one category. In MLNs, this rule may be *hard*, meaning that it behaves like a standard key constraint: in any possible world, each paper must be in at most one category. This rule may also be *soft*, meaning that it may be violated in some possible worlds. For example, in some worlds a paper may be in two categories. Soft rules also have *weights* that intuitively tell us how likely the rule is to hold in a possible world. In this example, F_1 is a soft rule and has weight 5. Roughly, this means that a fixed paper is e^5 times more likely to be in a single category compared to in 2 categories.[2]

Rules in MLNs are expressive and may involve data in non-trivial ways. For example, consider F_2:

$$\texttt{wrote}(x, p1), \texttt{wrote}(x, p2), \texttt{cat}(p1, c) \Rightarrow \texttt{cat}(p2, c) \quad (F_2)$$

Intuitively, this rule says that all the papers written by a particular person are likely to be in the same category. In this program, we believe that this rule is less likely to hold than F_1 as the weight of F_2 is $1 \leq 5$. Rules may also have existential quantifiers: F_4 says *"any paper in our database must have at least one author"*. It is also a hard rule, which is indicated by the infinite weight, and so no possible world may violate this rule. The weight of a formula may also be negative, which effectively means that the negation of the formula is likely to hold. For example, F_5 models our belief that none or very few of the unlabeled papers belong to 'Networking'. TUFFY supports all of these features.

Query Model. Given the data and the rules, a user may write arbitrary queries in terms of the relations. In TUFFY, the system is responsible for filling in whatever missing data is needed: in this example, the category of each unlabeled paper is unknown and so to answer the query the system must perform inference to understand what is the most likely label that they would take according to all available evidence.

2.2 Semantics of MLNs

We begin by describing the semantics of MLNs.[3] Formally, we first fix a schema σ (as in Fig. 1) and a domain of constants D. Given as input a set of formula $\bar{F} = F_1, \ldots, F_N$ (in *clausal form*[4]) with weights $w_1, \ldots, w_N$, they define a probability distribution over *possible worlds* (deterministic databases). To construct this probability distribution, the first step is *grounding*: given a formula F with free variables $\bar{x} = (x_1, \ldots, x_m)$, then for each $\bar{d} \in D^m$, we create a new formula $g_{\bar{d}}$ called a *ground clause* where $g_{\bar{d}}$ denotes the result of substituting each variable x_i of F with d_i. For example, for F_3 the variables are $\{p1, p2, c\}$: one tuple of constants is $\bar{d} = (\text{'P1'}, \text{'P2'}, \text{'DB'})$ and the ground formula $f_{\bar{d}}$ is:

$$\texttt{cat}(\text{'P1'}, \text{'DB'}), \texttt{refers}(\text{'P1'}, \text{'P2'}) \Rightarrow \texttt{cat}(\text{'P2'}, \text{'DB'})$$

Each constituent in the ground formula, such as `cat`('P1', 'DB') and `refers`('P1', 'P2'), is called a *ground predicate* or *atom* for short. In the worst case there are D^3 many ground clauses for F_3. For each formula F_i (for $i = 1, N$), we perform the above process. Each ground clause g of a formula F_i is assigned the same weight, w_i. So, a ground clause of F_1 has weight 5, while any ground clause of F_2 has weight 1. We denote by $G = (\bar{g}, w)$ the set of all ground clauses of $\bar{F}$ and a function w that maps each ground clause to its assigned weight. Fix an MLN $\bar{F}$, then for any possible world I define the cost of the world I to be

$$\text{cost}(I) = \sum_{g \in \bar{g}^+ : I \models \neg g} w(g) - \sum_{g \in \bar{g}^- : I \models g} w(g) \qquad (1)$$

where $\bar{g}^+$ and $\bar{g}^-$ are the sets of all ground clauses with positive and negative weights, respectively. Through cost, an MLN defines a probability distribution as:

$$\Pr[I] = Z^{-1} \exp\{-\text{cost}(I)\} \text{ where } Z = \sum_{J \in \text{Inst}} \exp\{-\text{cost}(J)\} \qquad (2)$$

To gain intuition, consider the special case where the range of w is $\{-1, 1\}$, then if a world I violates fewer rules than a

[1]The equality predicate is currently emulated by an explicit relation in TUFFY.

[2]Unfortunately, in MLNs, it is not possible to give a direct probabilistic interpretation of the weights [21]. In practice, the weights associated to formula are learned,which compensates for their non-intuitive nature. In this work, we do not discuss the algorithmics of learning.

[3]One can also give a procedural semantics of MLNs by compiling them into their namesake, a *Markov Network*. For completeness, we include this construction in §B.1.

[4]Clausal form is a disjunction of positive or negative literals. For example, the rule is $R(a) \Rightarrow R(b)$ is not in clausal form, but is equivalent to $\neg R(a) \vee R(b)$, which is in clausal form.

world J, then $\text{cost}(I) < \text{cost}(J)$; in turn, $\text{cost}(I) < \text{cost}(J)$ implies that I is *more likely* than J, i.e., $\Pr[I] > \Pr[J]$.

If the input MLN contains hard rules (indicated by a weight of $+\infty$ or $-\infty$), then we insist that the set of possible worlds (Inst) only contain worlds that satisfy every hard rule with $+\infty$ and violate every rule with $-\infty$. We also consider the situation when the MLN is provided with *evidence* which is a set of tuples such that each tuple t in the evidence is annotated as to whether t must or must not occur in any possible world; in turn, this restricts further the set of possible worlds (and changes Z). In practice, schemata have type information, and we can use this to remove nonsensical ground clauses, e.g., both attributes of `refers` are paper references, and so it is unnecessary to ground this predicate with another type, say person.

Example 1 If the MLN contains a single rule, then there is a simple probabilistic interpretation. Consider the rule R('a') with weight w. Then, Eq. 2 says that every possible world that contains R('a') has the same probability. Assuming that there are an equal number of possible worlds that contain R('a') and do not, then the probability that a world contains R('a') according to this distribution is $e^w/(e^w+1)$. In general, it is not possible to give such a direct probabilistic interpretation [21].

A *most likely world* is a possible world I^* that has highest probability among all possible worlds. The algorithmic goal of TUFFY is to find such a possible world; namely one with the lowest cost. This is a challenging task as it is NP-hard to find the most likely world [23], and so we need to resort to heuristic methods in practice. In fact, the reference implementation ALCHEMY constructs G, the set of ground formula, explicitly and then performs a search. As we will see, G may be huge (on the order of GiB), and so the reference implementation may exhaust memory and crash in many cases.

3. THE TUFFY SYSTEMS

We first describe BTUFFY that is a direct reimplementation of ALCHEMY, a state-of-the-art system for processing MLNs; the only major difference between ALCHEMY and BTUFFY is that an RDBMS is used for all data manipulation and memory management. We then analyze the BTUFFY prototype. Finally, we discuss CTUFFY, which builds on our experiences with BTUFFY.

3.1 bTuffy and Alchemy

There are two key algorithmic ideas in ALCHEMY [18, 29]: (1) *lazily* perform grounding and (2) *incrementally search* for the highest probability world using a heuristic taken from MAXSAT Solvers [11, 34].

Lazy Materialization. As ALCHEMY searches for a highest probability world, it lazily chooses a subset of clauses to ground; how ALCHEMY chooses this subset is easiest to understand by example. Consider the ground clause $g_{\bar{d}}$ of F_2 where $\bar{d}$ = ('Joe', 'P2' , 'P3', 'DB'). Suppose that in the evidence `wrote`('Joe', 'P3') is false, then $g_{\bar{d}}$ will be satisfied no matter how the other atoms are set ($g_{\bar{d}}$ is an implication). As a result, we can safely drop this ground clause from memory. Pushing this idea further, ALCHEMY works under the more aggressive hypothesis that most atoms will be false in the final solution, and in fact throughout the entire execution. To make this idea precise, call a ground clause *active* if it can be made unsatisfied by flipping zero or more active atoms, where an atom is active if its value *flips* at any point during execution, e.g., an atom is active if it is included in the initial world in the search, but is included on some later state; an atom is inactive otherwise. Observe that in the preceding example the ground clause $g_{\bar{d}}$ is not active. ALCHEMY keeps only *active ground clauses* in memory, which can be much smaller than the full set of ground clauses.

Both ALCHEMY and BTUFFY maintain the set of active ground clauses and atoms throughout execution. In ALCHEMY, all structures are stored in main memory. BTUFFY stores the active atoms of each predicate in a separate relation: for each predicate $P(\bar{A})$ in the input MLN, BTUFFY creates a relation R_P(tid, $\bar{A}$, truth, state) where *tid* is an auto-generated row id, $\bar{A}$ is the tuple of arguments of the predicate P, *truth* is a Boolean-valued attribute that represents the truth assignment, and *state* is an enumerated type where the value is one of query, evidence, active, or default. To store the ground clauses, ALCHEMY uses arrays, whereas BTUFFY constructs a single *clause table* that holds all ground clauses. The schema of the clause table is C(rowid, tids, weight, nSat). Each tuple $t \in C$ corresponds to a ground clause; $t.rowid$ is a unique identifier, $t.tids$ is an array that stores (possibly negated) ids of the atoms in this ground clause, and $t.weight$ is its weight. The attribute $nSat$ is the number of currently satisfied atoms, which is used by the inference algorithm that we explain below.

WalkSAT *Search.* The highest probability world is the one in which the cost is the lowest. Since we have stored the grounded clauses in clausal form, we have a weighted MAXSAT problem.[5] MAXSAT is NP-hard (even to approximate [7, 37]), and so we must resort to heuristics. ALCHEMY (and so BTUFFY) run a fixed number of iterations of a greedy local-search procedure called WALKSAT [34]. For simplicity, we assume that all clause weights are positive. Each iteration of the greedy local-search procedure first picks an unsatisfied clause (whose $nSat$ value is 0), and then the procedure satisfies that clause. This is done by flipping an atom (i.e., changing its value from true to false or vice versa) in the clause, and the procedure chooses (stochastically) between two approaches: with some probability, say $\frac{1}{2}$, we perform a *greedy step* and select the atom in the clause that would reduce the cost most significantly if its value is flipped (the amount of change is called the δ-cost). Otherwise, with probability $\frac{1}{2}$, we select a random atom uniformly. This second step is designed to avoid the local minima that we would get stuck in if the algorithm only contained the pure greedy steps. If the selected atom a is not active, then we set a to active; in turn, this may generate new ground clauses, which are then added to the clause table. Then, we update the $nSat$ attribute of clauses that contain this atom, and finally the δ costs of atoms contained by these clauses. The pseudocode of WALKSAT can be found in §C.1.

[5] Given a CNF Formula with disjuncts $d_1, \ldots, d_N$ and weights $w_1, \ldots, w_N$, the MAXSAT problem is to find the assignment that maximizes the weights of the satisfied disjuncts. This is a generalization of SAT, a canonical NP-Hard problem.

3.2 Analyzing bTuffy

The first lesson that we learned from bTuffy was not surprising: implementing Alchemy in an RDBMS allows us to process much larger data sets. A second (and more surprising) lesson is that Alchemy does a large amount of typical relational operations, and that these operations are a large fraction of Alchemy's execution time. The third lesson that we learn is that RDBMSes are not well-suited to the search algorithm of Alchemy (a typical AI search program).

bTuffy scales when Alchemy does not. Alchemy is confined to main memory and so it is no surprise that when main memory is exhausted, Alchemy crashes. On the entity resolution benchmark available from Alchemy's website, Alchemy crashes on a machine with 4GB of RAM without completing initialization. In contrast, bTuffy completes the initialization process in approximately 11 minutes; after this, bTuffy continues to perform inference without crashing.

MLNs perform relational operations. During the initialization phase, Alchemy must create and manipulate a large chunk of data using relational operations. In bTuffy, it is relatively straightforward to reproduce Alchemy's initialization strategy using standard SQL and features found in every major commercial (and open-source) database.[6] In Alchemy, the relational operations are a major bottleneck: using the default settings for Alchemy on a relational classification task, 96% of the time to perform inference is spent simply initializing.

To see why initialization is so costly, we observe that in Alchemy, the main-memory analog of the clause table is built using hand-coded C++ that essentially implements the SQL program of bTuffy. Alchemy is not naïve: it implements several heuristics to choose the join ordering and pushes down predicates as much as possible (this section of the Alchemy code is relatively complex). Still in bTuffy, since the initialization is expressed in SQL, the RDBMS optimizer selects a low-cost evaluation strategy automatically. The difference between the two approaches is dramatic: on a relational classification task (**RC**), Alchemy takes over 1 hour to initialize, while bTuffy initializes in only 23 seconds.

Drilling deeper, we observe that the feature of the relational optimizer that is most helpful to bTuffy is automatically selecting the correct join algorithm to use and pushing down predicates. In particular, for the relational classification task, Alchemy effectively uses a naive strategy to process the join implicit in grounding the formula; in contrast, the PostgreSQL optimizer computes a significantly more sophisticated plan: it pushes down projections, thus pruning the intermediate state considerably. In contrast, Alchemy uses a hard-coded nested loops strategy. To support this point, we conducted a lesion study in §D, which shows that it's the lack of standard RDBMS join algorithms such as sort-merge join and hash join that kills Alchemy.

From this we conclude that the RBDMS is able to provide substantial value to help MLNs with their data manipulation.

[6]We include an example query and its translation in §C.2.

Search Algorithms in an RDBMS. Not everything is so rosy for bTuffy: the search algorithm of Alchemy, called WalkSAT, is difficult to implement using an RDBMS and slow. WalkSAT requires that bTuffy manipulate many fine-grained data structures, e.g., flipping an atom requires us to find all those clauses which are now unsatisfied; in turn, this affects the δ cost of (many) other atoms; in turn, this affects which atoms we pick, etc. Each of these fine-grained manipulations requires a SQL query, and the sheer number of SQL queries is astonishing: On the information extraction dataset, we need to issue more than 625,000 SQL queries to complete it.

A second issue is that the WalkSAT algorithm is not I/O aware. Conventional wisdom dictates that creating indexes would cut down on the number of IO operations, but we found that the opposite is true: in all cases we experimented with indexing causes the overall runtime to slow down. To see why, consider that WalkSAT needs to randomly select a clause that is unsatisfied, yet indexing on whether a clause is satisfied would require costly updates to the index. Even if this problem is bypassed, we need to recompute the δ cost values, the $nSat$ values, etc. As a result, even a single flip requires several scans over the data. So, while Alchemy achieves a flip rate of 330,000 flips per second on the **IE** dataset, bTuffy limps along on the same dataset at a flip rate of 17 flips per second.[7] Reducing the number of queries, making the algorithm both more IO friendly, and improving the effective flip rate is a key design goal for our improved implementation, cTuffy, that we discuss next.

3.3 The cTuffy System

In cTuffy, the main ideas are to exploit *bulk loading* and *set-at-a-time processing* whenever possible. The first algorithmic idea is to use a MAXSAT algorithm that performs more than one flip per iteration. The effects can be dramatic, to find the lowest cost solution for the entity resolution dataset **ER** cTuffy takes 680 seconds, while Alchemy takes over 7 hours and bTuffy over 20 hours. We explore this point more fully in the next section. Our second algorithmic technique takes advantage of bulk-loading by precomputing the atoms that will be needed by Alchemy, and so allows a more eager strategy; thereby cutting down on the large number of update statements to improve performance. Our third algorithmic technique speeds up initialization by recognizing structure in the input program called *templates*. In particular, by grouping together many tiny updates during initialization we are able to achieve over a factor of 15 improvements in initialization on the information extraction dataset (from 159 seconds to 9 seconds).

Set-at-a-time MAXSAT. Our starting observation is that in Alchemy on each iteration of the greedy local search, the state of at most one atom changes. Thus, if our solution must change a large number of atoms, cTuffy must issue a large number of SQL queries, since each flip affects at most one atom and each flip is implemented by at least one SQL queries. Instead, in cTuffy we will flip many atoms simultaneously and thus take advantage of *set-at-a-time processing*. To select the atoms to flip, we call an atom "good" if changing its value alone would decrease the cost. At each step of our search, we flip a random fraction (e.g.,

[7]We conducted a query throughput experiment, and the results is presented in §C.3.

Algorithm 1 cTuffy Inference Algorithm

Input: A: a set of atoms
Input: C: a set of weighted ground clauses
Input: `ModeThreshold`, `Greediness`, `LocalNum`
Input: `MaxFutileSteps`, `MaxRestarts`
Output: σ^*: a truth assignment to A

```
lowCost ← +∞, σ* ← the all-false assignment
for try = 1 to MaxRestarts do
  σ ← a random truth assignment to A
  tryLowCost ← +∞, futileSteps ← 0
  while futileSteps < MaxFutileSteps do
    numGood ← the number of good atoms
    // flipping atoms changes σ
    if numGood > ModeThreshold then
      flip each good atom with probability Greediness
    else if numGood > 0 then
      randomly choose LocalNum good atoms and flip
    else
      randomly choose LocalNum atoms and flip
    cost ← the cost as defined in §2
    if cost < lowCost then
      lowCost ← cost, σ* ← σ
    if cost < tryLowCost then
      futileSteps ← 0, tryLowCost ← cost
    else
      futileSteps ← futileSteps + 1
```

0.5) of all good atoms, so that we can get close to the best assignment very quickly. Once the number of good atoms is lower than a certain threshold, cTuffy enters the local search phase, in which we flip only a small number (e.g., 1 or 5) of good atoms at each step. This allows us to explore the space in a more fine-grained manner and find potentially higher-valued solutions. Pseudocode of this algorithm is listed in Algorithm 1. On all of our four benchmarks, we found that cTuffy consistently returns better solutions than Alchemy (§4.1).

Computing the Closure. The main idea of this optimization is to trade inserts that occur during Alchemy's search procedure, for inserts that occur before its procedure; then, we can bulk-load the inserts and so perform them more efficiently. To do this, we need to compute those atoms that will become active during Alchemy's execution, which we call the *closure*. Initially, we begin with an empty set of active atoms $P_0 = \emptyset$, assuming that all non-evidence atoms are false. Then, we ground only those clauses that (1) are not satisfied by evidence and (2) are not satisfied by inactive atoms. From this we obtain the a set of active ground clauses, G_1; adding into P_0 the atoms that appear in some ground clause from G_1, we obtain P_1. We then repeat with P_1 as the active set of atoms. Continuing in this way, we define a set of atoms $P_1, P_2, \ldots$. We show the following in the appendix (§C.4):

Proposition 3.1. *Fix an* MLN *and let* $P_1, \ldots, P_t, \ldots$ *be the sets of ground atoms defined above. On any run of* Alchemy*, let* A_t *be the set of active ground atoms after* t *flips, then* $A_t \subseteq P_t$.

Essentially, this proposition says that our algorithm computes a superset of the active grounded clauses. In our experiments, P_t converges rapidly, that is, $P_t = P_{t+1}$ for a small values of t (say $t = 4$). As a result, in our current prototype, we simply run the above procedure until convergence. Pseudocode for this algorithm is in §C.4. Then, we bulk-load P_t and its corresponding set of ground clauses G_t.

	LP	IE	ER	RC
#relations	22	18	10	4
#rules	94	1,024	3,812	15
#entities	302	2,608	510	50,710
#evidence tuples	731	255,532	676	429,731
#query atoms	4,624	336,670	15,876	10,000
#active atoms	2,526	6,819	20,205	9,860
clause table size	5.2 MB	0.6 MB	164 MB	4.8 MB
database size	31 MB	111 MB	2.1 GB	95 MB
Alchemy RAM	411 MB	206 MB	3.5 GB	2.8 GB

Figure 2: Dataset statistics

Bulk loading Templates. Sometimes the rules in an MLN program can be grouped by patterns, where each group can be summarized by a template of the form:

$$\texttt{token}(+word, i, c), \neg\texttt{isDigit}(+word) \Rightarrow \texttt{inField}(c, +field, i)$$

Each individual rule in the group has its own weight, and can be obtained by substituting constants in place of the variables preceded by '+' in the formula. A group like this can be very large if such variables take on many possible values (e.g., *word* has thousands of possible assignments); grounding all of them requires thousands of SQL queries, which would incur substantial overhead. cTuffy addresses this by introducing a table for each such template with the schema $\texttt{ruleSpec}(word, field, weight)$. Thus, grounding can be done on a template-by-template basis, greatly reducing the time of initialization (§4.2).

4. EXPERIMENTS

We empirically validate that the use of an RDBMS in Tuffy provides scalability and orders of magnitude faster query processing speed compared to Alchemy; that the performance of cTuffy dominates both Alchemy and bTuffy; and that the optimizations of bulk loading and set-at-a-time processing in cTuffy are the keys to its success.

Experimental Setup. Alchemy and Tuffy are implemented in C++ and Java, respectively. The RDBMS used by Tuffy is PostgreSQL 8.4. We ran all experiments on an Intel Core2 6600 at 2.4GHz with 3.7 GB of RAM running Red Hat Enterprise Linux 5. There are several parameters required by Alchemy and the Tuffy systems, and we set them as follows: Alchemy and bTuffy run 10,000 flips/try. During global search of cTuffy, the fraction of selected good atoms is 0.5; we analyze cTuffy's sensitivity to this parameter in §D. cTuffy enters the local search phase when the number of good atoms is less than 100. During local search of cTuffy, up to 5 good atoms are chosen at each step.

4.1 An Empirical Study of Tuffy and Alchemy

We first describe a set of experiments that is critical to the analysis of bTuffy in §3.2, and is a useful yardstick to measure the performance advantages of cTuffy. We run Alchemy, bTuffy, and cTuffy on four benchmark tasks with real datasets. Three of the datasets and MLNs are taken directly from the Alchemy website [5]: *Link Prediction* (**LP**), given part of the administrative database in the University of Washington CS department, the goal is to

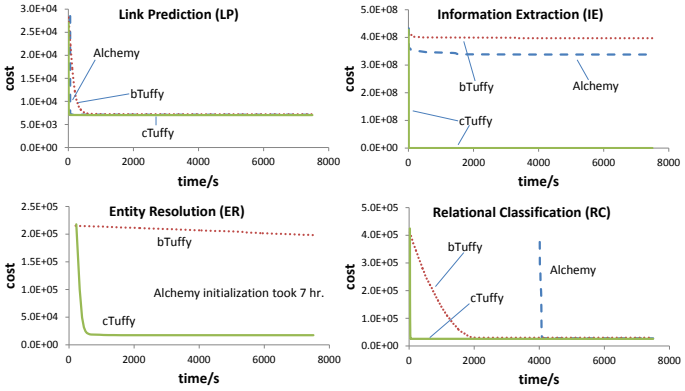

Figure 3: Performance of Alchemy and Tuffy

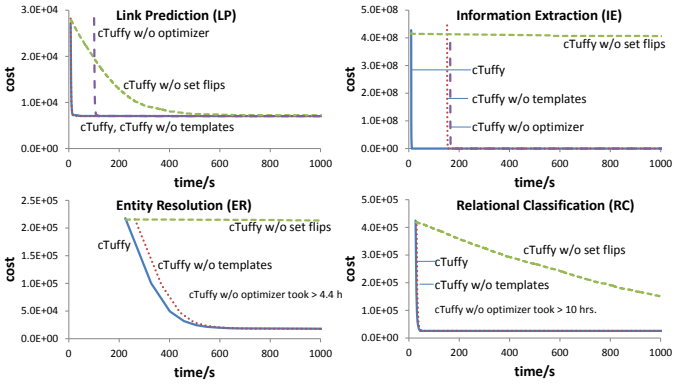

Figure 4: Component contributions

predict student-adviser relationships; *Information Extraction* (**IE**), given a set of Citeseer citations, the goal is to extract from them structured records based on some domain-specific heuristics[8]; and *Entity Resolution* (**ER**), which is to deduplicate citation records based on word similarity. These tasks have been extensively used in many other works on MLNs [15, 18, 27–29]. The last task, *Relational Classification* (**RC**), performs classification on the Cora dataset [12]; **RC** contains all the rules in Figure 1. Statistics for these four datasets are listed in the first five rows of Figure 2.

For each combination of dataset, task, and system, we plot in Fig. 3 a graph where on the x axis is the length of time the approach runs, and on the y axis is the cost of the best solution that the approach has found to that point. Informally, one approach dominates another approach if it achieves an equal or lower cost at each point. Immediately, we can see the most striking observation is that on all the four tasks, cTuffy dominates both Alchemy and bTuffy; that is, it more quickly achieves every single cost. Moreover, on **ER** and **RC**, cTuffy and bTuffy are able to reduce the initialization time by two orders of magnitude compared to Alchemy (7 hours to 3 minutes on **ER**; 1.1 hours to 0.5 minute on **RC**). On these two datasets, we see that cTuffy significantly improves upon bTuffy by retaining its scalability and fast initialization while greatly speeding up inference with the set flip algorithm.

We also verified that the cost-based RDBMS optimizer consistently found sophisticated query plans for the grounding process, which is in contrast to Alchemy's hard-coded nested loop. As a side effect, Tuffy is much more space-efficient than Alchemy since the intermediate data are much smaller. For example, on **RC**, the size of the clause table is only 95 MB in both cTuffy and bTuffy, whereas the RAM footprint of Alchemy is 2.8 GB (Fig. 2). In terms of inference, the difference between cTuffy and bTuffy is shown dramatically on the most challenging task, **ER**, where cTuffy can find a solution within minutes, whereas bTuffy keeps limping at an extremely slow speed. On the other hand, although the clause table of **IE** is the smallest, bTuffy and Alchemy still seem to have great difficulties reaching a solution. This is due to the fact that the WalkSAT algorithm is vulnerable to rules with disparate weights, as is the case with the MLN of **IE**, which contains both rules with a weight 10^6 and rules with a weight 1.

We now investigate the effects of each optimization systematically on these results using a lesion study on cTuffy.

Lesion Study: Individual Optimizations. We run cTuffy together with three variants on the previous four tasks: 1) cTuffy with the set-flip algorithm disabled; 2) cTuffy with template bulk loading disabled; and 3) cTuffy with the database optimizer disabled. The last one is obtained by limiting the RDBMS to use nested-loop join algorithm and forcing the join orders to be the same as in Alchemy.[9] As shown in Fig. 4, without set-flip inference, cTuffy performs very poorly in searching for a solution. Without template bulk loading, the initialization time suffers substantially on datasets with a large number of rules such as **IE** and **ER**. Lastly, on all four tasks, it's clear that the optimizer plays a critical role in speeding up the initialization phase of cTuffy. These results suggest that the cTuffy optimizations of bulk loading and set-oriented data processing directly contribute to its success.

4.2 Scalability Experiments

In this section, we discuss scalability. In §D, we discuss extended experiments that we craft to explore the limits of Tuffy and Alchemy.

Scalability: Data Size. We compare the scalability of all three systems with respect to the data size and the query size. We select **RC** as the testbed. In Figure 5, we fix the query size (i.e., the number of unlabeled papers) to be 1000, and vary the evidence to be random fractions or copies of the original dataset. (**RCx** means the size of the evidence is **x** times that of **RC**; e.g., **RC2** has twice as many evidence as **RC**.) The results show that while bTuffy and cTuffy are very robust to the data size, Alchemy does not scale well as the the data size grows.

Scalability: Query Size. In a second group of experiments, we fix the dataset to be **RC**, but vary the size of the query between 1000 and 4000. (**RC Qx** means the size of the query is **x**; e.g., **RC Q1000** is equivalent to **RC**, and the query of **RC Q2000** contains 2000 papers.) As

[8]Currently, Tuffy does not directly support hard rules; the hard rules are transformed into practically equivalent rules with very high weights

[9]We also conducted a lesion study that separates the contributions of these two aspects, see §D.

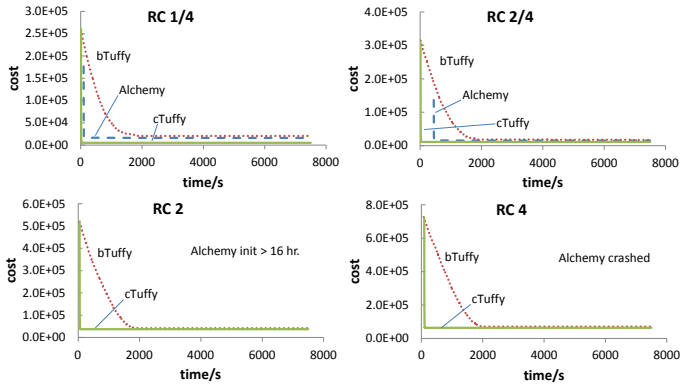

Figure 5: Scalability in data size

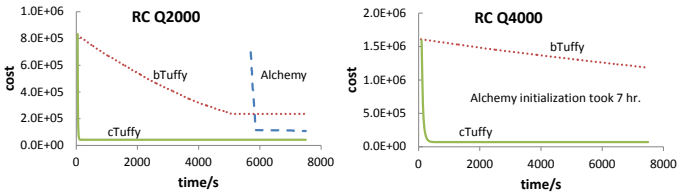

Figure 6: Scalability in query size

shown in Figure 6, cTuffy is able to find the best solution within minutes even when Alchemy took hours to initialize. On the other hand, although the initialization time of bTuffy was almost unchanged, its inference performance is very sensitive to the query size. This further suggests that bulk inference in cTuffy is superior to WalkSAT.

5. CONCLUSION AND FUTURE WORK

Motivated by a staggeringly large set of applications, we propose Tuffy that pushes MLN inference inside an RDBMS. We find that MLN inference uses many "relational-like" operations and that these operations are a substantial bottleneck in MLN inference. To alleviate this bottleneck, we propose cTuffy, which in addition to using an RDBMS modified the state-of-the-art search algorithms to be *set-at-a-time*. We believe that we are the first to push search algorithms inside an RDBMS, and that our prototype cTuffy demonstrates that this is a promising new direction for database research to support increasingly sophisticated statistical models. As future work, we plan to study whether main memory databases can be used in conjunction with RDBMSes to provide more efficient implementation of search procedures.

6. REFERENCES

[1] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, 2008.

[2] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 2008.

[3] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[4] A. Deshpande and S. Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD Conference*, 2006.

[5] P. e. Domingos. `http://alchemy.cs.washington.edu/`.

[6] R. Fagin, J. Y. Halpern, and N. Megiddo. A logic for reasoning about probabilities. 1990.

[7] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 1995.

[8] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[9] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *ICDE*, 2008.

[10] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. *The Satisfiability Problem: Theory and Applications*, 1997.

[11] L. Kroc, A. Sabharwal, C. P. Gomes, and B. Selman. Integrating systematic and local search paradigms: A new strategy for maxsat. In *IJCAI*, 2009.

[12] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval Journal*, 2000. www.research.whizbang.com/data.

[13] B. L. Milenova, J. S. Yarmus, and M. M. Campos. Svm in oracle database 10g: removing the barriers to widespread adoption of support vector machines. In *VLDB '05*.

[14] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. 1988.

[15] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI '07*.

[16] H. Poon and P. Domingos. Joint unsupervised coreference resolution with Markov Logic. In *EMNLP '08*.

[17] H. Poon and P. Domingos. Unsupervised semantic parsing. In *EMNLP '09*.

[18] H. Poon, P. Domingos, and M. Sumner. A general method for reducing the complexity of relational inference and its application to MCMC. *AAAI-08*.

[19] C. Re, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.

[20] C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD Conference*, 2008.

[21] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 2006.

[22] S. Riedel and I. Meza-Ruiz. Collective semantic role labelling with markov logic. In *CoNLL '08*.

[23] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 1996.

[24] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[25] P. Sen, A. Deshpande, and L. Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 2009.

[26] P. Sen, A. Deshpande, and L. Getoor. Prdb: managing and exploiting rich correlations in probabilistic databases. *J. VLDB*, 2009.

[27] J. Shavlik and S. Natarajan. Speeding up inference in Markov logic networks by preprocessing to reduce the size of the resulting grounded network. In *IJCAI-09*.

[28] P. Singla and P. Domingos. Entity resolution with Markov logic. In *ICDE '06*.

[29] P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *AAAI '06*.

[30] P. Singla, H. Kautz, J. Luo, and A. Gallagher. Discovery of social relationships in consumer photo collections using Markov Logic. In *CVPR Workshops 2008*.

[31] C. Stiller, G. Farber, and S. Kammel. Cooperative Cognitive Automobiles. In *2007 IEEE IVS*.

[32] S. Tran and L. Davis. Event Modeling and Recognition Using Markov Logic Networks. In *ECCV '08*.

[33] D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 2008.

[34] W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, 2004.

[35] F. Wu and D. S. Weld. Automatically refining the wikipedia infobox ontology. In *WWW '08*, 2008.

[36] L. Xiao, J. Gerth, and P. Hanrahan. Enhancing visual analysis of network traffic using a knowledge representation. In *VAST '07*.

[37] M. Yannakakis. On the approximation of maximum satisfiability. *J. Algorithms*, 1994.

APPENDIX

A. EXTENDED RELATED WORK

The idea of using the stochastic local search algorithm WalkSAT to find the most likely world was due to Kautz et al. [10]. Singla and Domingos [29] extends WalkSAT with lazy materialization in the context of MLNs, resulting in an algorithm called LazySAT that is used by both Alchemy and bTuffy. The idea of ignoring ground clauses that are satisfied by evidence is highlighted as an effective way of speeding up the MLN grounding process in Shavlik and Natarajan [27], which formulates the grounding process as nested-loop joins and provides ad hoc heuristics to approximate the optimal join order.

In designing an alternative MAP solver to WalkSAT, Riedel [39] also observed that the grounding process can be cast into SQL queries in a straightforward manner. He correctly pointed out that "Database technology is optimised for this setting."

B. SUPPORTING MATERIAL FOR §2

B.1 The Resulting Markov Network

A *Boolean Markov network* (or *Boolean Markov Random Field* is a model of the joint distribution of a set of Boolean random variables $\bar{X} = (X_1, \ldots, X_N)$. It is defined by two related parts: first, a graph structure $G = (X, E)$; let $\mathcal{C}$ denote the set of cliques in G, then for each $C \in \mathcal{C}$ there is a *potential function* denoted ϕ_C, which is a function from the values of the set of variables in C to non-negative real numbers. Together, these two pieces define a joint distribution $\Pr(\bar{X} = \bar{x})$ as follows:

$$\Pr(\bar{X} = \bar{x}) = \frac{1}{Z} \prod_{C \in \mathcal{C}} \phi_C(\bar{x}_C)$$

where $\bar{x} \in \{0,1\}^N$, Z is a normalization constant and $\bar{x}_C$ denotes the values of the variables in C.

Fix a set of constants $C = \{c_1, \ldots, c_M\}$. An MLN defines a Boolean Markov Network as follows: for each possible grounding of each predicate, create a node (and so a Boolean random variable). For example, there will be a node `refers`$(p1, p2)$ for each pair of papers $p1, p2$. For each formula F_i we ground it in all possible ways, then we create a clique C that contains the nodes corresponding to all terms in the formula. For example, the key constraint then creates cliques for each paper and all of its potential categories. This feature ϕ_i is 1 when its true and 0 when its false; the weight of this feature is w_i. We refer to this graph as the *ground network*.

Once we have the ground network, our task reduces to inference in Markov models.

C. SUPPORTING MATERIAL FOR §3

C.1 The LazySAT Algorithm

The inference algorithm used by Alchemy and bTuffy is called LazySAT [29]. For completeness, we include its pseudocode here (Algorithm 2). For simplicity, we assume that all clause weights are positive.

Algorithm 2 The LazySAT Algorithm

Input: A: an initial set of atoms
Input: C: an initial set of weighted ground clauses
Input: `MaxFlips`, `MaxRestarts`
Output: σ^*: a truth assignment to A

```
lowCost ← +∞, σ* ← the all-false assignment
for try = 1 to MaxRestarts do
  σ ← a random truth assignment to A
  for flip = 1 to MaxFlips do
    if cost = 0 then
      return σ* // found a solution, halt
    pick a random c ∈ C that's unsatisfied
    rand ← a random float between 0.0 and 1.0
    if rand ≤ 0.5 then
      // a random step
      atomToFlip ← a random atom in c
    else
      // a greedy step
      atomToFlip ← an atom in c with lowest δ-cost
    if atomToFlip is inactive then
      A ← A ∪ {atomToFlip}
      C ← C∪ ground clauses activated by atomToFlip
    flip atomToFlip
    update the cost, as defined in §2
    if cost < lowCost then
      lowCost ← cost, σ* ← σ
```

C.2 An Example SQL Query populating the Clause Table

Consider the formula F_3 in Fig. 1. Suppose that the actual schemata of `cat` and `refers` are `cat`(tid, paper, category, truth, state) and `refers`(tid, paper1, paper2, truth, state), respectively. Given the evidence and the current set of active atoms – as indicated by the "truth" and "state" attributes – we can ground the active clauses for this formula using the following SQL query.

```
SELECT -t1.tid, -t2.tid, t3.tid
FROM cat t1, refers t2, cat t3
WHERE (t1.state=ACTIVE OR
       (t1.state=EVIDENCE AND t1.truth))
AND (t2.state=ACTIVE OR
     (t2.state=EVIDENCE AND t2.truth))
AND (t3.state=ACTIVE OR
      t3.state=INACTIVE OR
     (t3.state=EVIDENCE AND NOT t3.truth))
AND t1.paper=t2.paper1
AND t1.category=t3.category
AND t2.paper2=t3.paper
```

Note that the tids of t1 and t2 are negated because the corresponding predicates are negated in the clausal form of F_3.

C.3 SQL Throughput in bTuffy

We run bTuffy on the datasets described in §4 and measure the number of SQL queries along the time. In Fig. 7, the monotonically increasing (blue) curve indicates the cumulative number of SQL queries that bTuffy has issued over time; the other (green) one is its derivative, i.e., the RDBMS's query processing speed. Clearly, as the size of the clause table increases, RDBMS's throughput deteriorates rapidly. On the most challenging task, **ER**, Postgres cannot even finish one query (a table scan in effect) within a

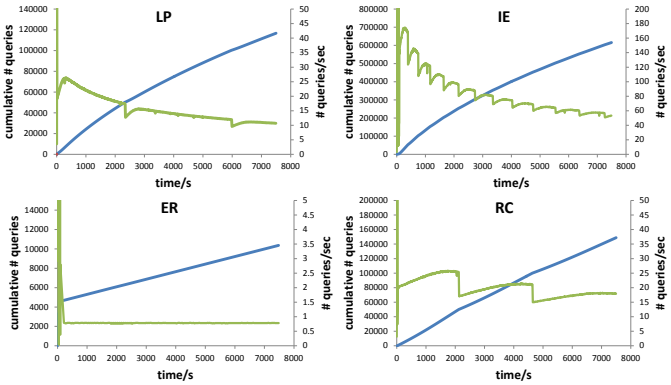

Figure 7: SQL throughput in bTuffy

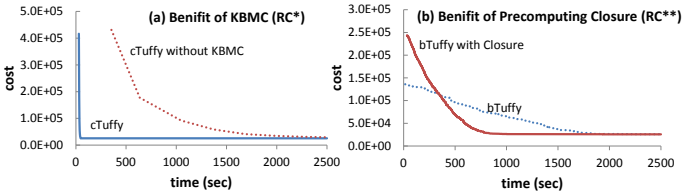

Figure 8: KBMC and Closure precomputation

	IE-t	RC-t
with template loading	9 sec	19 sec
without template loading	16 min	94 min

Figure 9: Initialization time of cTuffy

second. An interesting phenomenon from the graph is how the SQL throughput changes over time. We found that it has to do with Postgres's mandatory multi-version concurrency control (MVCC) component. This further motivates us to consider in-memory databases as a direction of future work.

C.4 Computing the Lazy Closure

For simplicity, we present a relatively naive algorithm to compute the lazy closure as defined in §3.3 (Algorithm 3). We can naturally derive a semi-naive algorithm, in the same spirit as semi-naive evaluation of Datalog. cTuffy implements a semi-naive algorithm.

Algorithm 3 Precomputing LazySAT Closure

Input: a set of MLN rules with a domain of constants
Input: a database of evidence
Output: A_c: the closure of active atoms
Output: G_c: the closure of active ground clauses

$A_c \leftarrow \emptyset$
`converged` $\leftarrow$ **false**
while `converged` = **false do**
  `G` $\leftarrow$ ground clauses of all rules that are
    not satisfied by the evidence,
    and not satisfied by inactive atoms (those not in A_c)
  `savers` $\leftarrow$ atoms contained in `G`
  if `savers` $\subseteq A_c$ **then**
    `converged` $\leftarrow$ **true**
  else
    $A_c \leftarrow A_c \cup$ `savers`
    $G_c \leftarrow G_c \cup$ `G`

C.5 Proof of The Closure Proposition

Proof of Proposition 3.1. Given an MLN, the set of active ground clauses is a function G of the current set of active atoms P. Furthermore, from the definition of G, it's not hard to see that it is a monotone function: for any $P \subseteq P'$, we have $G(P) \subseteq G(P')$. Let G_x be a set of ground clauses; define $E(G_x)$ to be the set of atoms contained by some ground clause in G_x. Then the closure algorithm can be expressed by the initial condition $P_0 = \emptyset$ and the following formula:

$$P_{k+1} = P_k \cup E(G_{k+1}) = P_k \cup E(G(P_k)).$$

In Alchemy, the initial set of active atoms is $A_0 = P_0 \subseteq P_0$. Then for $k \geq 1$, $A_{k+1} = A_k \cup a_{k+1}$ where a_{k+1} is the ground atom chosen to be flipped at iteration $k+1$. Since $a_{k+1} \in E(G(A_k))$, we have $A_{k+1} \subseteq A_k \cup E(G(A_k))$. Suppose that $A_k \subseteq P_k$, then by monotonicity of the functions G and E, $A_{k+1} \subseteq P_k \cup E(G(P_k)) = P_{k+1}$. By induction, the claim holds. □

D. EXTENDED EXPERIMENTS

We discuss several lesion studies to evaluate extensions to our framework.

Lesion Study: KBMC. cTuffy implements knowledge-based model construction (KBMC) [42], a standard way of pruning facts and rules that are irrelevant to the query at hand. For the purpose of testing, we created a dataset **RC*** by adding into **RC** a relation `affiliation` and a rule that's not related to `cat`. As shown in Fig. 8(a), by restricting cTuffy to relevant groundings, KBMC can drastically speed up both initialization and inference.

Lesion Study: Lazy Closure. In §3.3, we described an algorithm to precompute the closure of active groundings needed by cTuffy. As it's very hard to separate this from the cTuffy inference algorithm, we conducted a lesion study on bTuffy instead. To ensure that the initial set of active atoms does not equal the closure, we removed a non-essential rule from the MLN of **RC**, resulting in a task called **RC****. As shown in Fig. 8(b), by consolidating all the fine-grained incremental grounding processes, bTuffy augmented with closure precomputation can find a solution within half the time of bTuffy.

Lesion Study: Template Bulk Loading. To single out the benefit of template bulk loading, we measured the initialization time of cTuffy on two more datasets with this optimization enabled and disabled, respectively. The first dataset, **IE-t**, is obtained by ten-folding the English words in the **IE** MLN (from thousands to tens of thousands). The second one, **RC-t**, is derived from **RC** by replacing x with $+x$ in rule F_2 and then instantiating it into individual rules, one for each Person constant.[10] As shown in Fig. 9, template bulk loading can speed up initialization by orders of

[10]Intuitively, this allows us to model the breadth of study of individual researchers.

magnitude.

Lesion Study: RDBMS optimizer components. To investigate which part of the RDBMS optimizer is the deciding factor of TUFFY's vastly improved initialization speed over ALCHEMY, we designed a lesion study to compare two aspects of the optimizer: join algorithms and join ordering. Using the datasets as described in §4, we run cTUFFY and two variants: 1) cTUFFY with nested loop, where we force Postgres to use nested-loop joins only; and 2) cTUFFY with fixed order, where we constrain Postgres to follow the same join order as ALCHEMY.

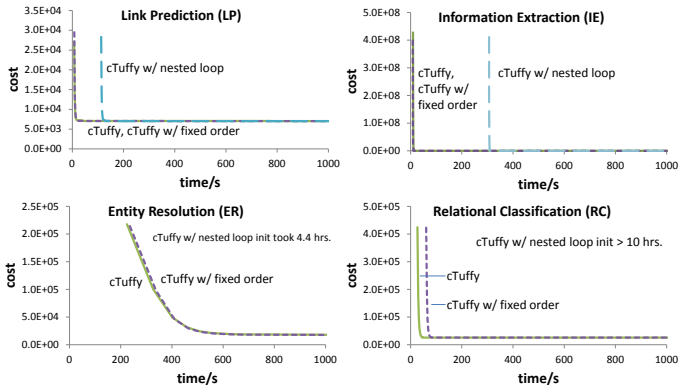

Figure 10: Effects of join algorithm v. join order

As shown in Fig. 10, while the join order has only minor impacts, not being able to select better join algorithms (other than nested loop) virtually deprived the advantages of the optimizer. Although a better join order does significantly improve the performance in certain situations – as shown in [27] – the result of this lesion study suggests that tweaking the join order alone could still result in evaluation plans that are far from optimal. Thus, it is crucial to jointly consider join order and join algorithms – an expertise of RDBMS optimizers.

Sensitivity Analysis. We empirically test cTUFFY's sensitivity to its degree of "greediness" in global search; this corresponds to the fraction of good atoms to be flipped at each step. We vary this parameter between 0.3 and 0.9, and ran cTUFFY on two tasks. As shown in Figure 11, the performance of cTUFFY keeps improving as we make it greedier and greedier. We suspect that such a trend is due to the structural regularity of the input MLN and plan to further study cTUFFY's inference algorithm.

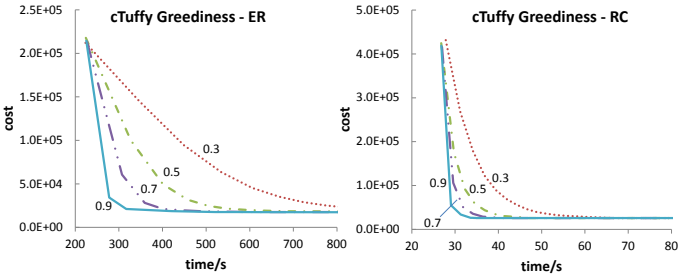

Figure 11: Sensitivity analysis of cTuffy

E. REFERENCES

[38] H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. *The Satisfiability Problem: Theory and Applications*, 1997.

[39] S. Riedel. Improving the accuracy and efficiency of map inference for markov logic. In *UAI*, 2008.

[40] J. Shavlik and S. Natarajan. Speeding up inference in Markov logic networks by preprocessing to reduce the size of the resulting grounded network. In *IJCAI-09*.

[41] P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *AAAI '06*.

[42] M. Wellman, J. Breese, and R. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 2009.