Felix: Scaling Inference for Markov Logic with an Operator-based Approach

Feng Niu   Ce Zhang   Christopher Ré   Jude Shavlik

University of Wisconsin-Madison
{leonn, czhang, chrisre, shavlik}@cs.wisc.edu

August 1, 2011

Abstract

We examine how to scale up text-processing applications that are expressed in a language, Markov Logic, that allows one to express both logical and statistical rules. Our idea is to exploit the observation that to build text-processing applications one must solve a host of common subtasks, e.g., named-entity extraction, relationship discovery, coreference resolution. For some subtasks, there are specialized algorithms that achieve both high quality and high performance. But current general-purpose statistical inference approaches are oblivious to these subtasks and so use a single algorithm independent of the subtasks that they are performing. The result is that general purpose approaches have either lower quality, performance, or both compared to the specialized approaches. To combat this, we present FELIX. In FELIX programs are expressed in Markov Logic but are executed using a handful of predefined operators that encapsulate the specialized algorithms for each subtask. Key challenges are that FELIX (1) must find these subtasks in the input program, (2) must have an algorithm that allows the specialized algorithms for each subtask to communicate, (3) must ensure that the entire framework is efficient. To address these challenges, FELIX's contributions are: (1) an architecture and semantics for our operator-based approach, (2) a compiler that discovers operators automatically, and (3) a cost-based optimizer for data movement between operators. We apply FELIX to information extraction, and we demonstrate that while prior statistical approaches crash, FELIX is able to complete (often an order of magnitude more efficiently). In turn, FELIX can achieve higher quality than state-of-the-art IE approaches on three real-world datasets.

1 Introduction

There are myriad of data sources that are valuable to developers, but using these data sources in applications is difficult. A key reason is that the diversity of structure and content of these data sources makes it infeasible for a developer to perfectly understand each data source, e.g., text on the Web or inside enterprises, community data from Wikipedia[1], or structured data from Freebase.[2] The number and diversity of sources available to a developer is continually increasing, which suggests that building applications with such data sources is an increasingly important problem.

Building an application with such data is a challenging task. A key challenge is that a developer has only a partial understanding about the structure and format of the data, and so the statements they can make about the data are not fully correct, but instead are only likely to be correct. For example, *"if the set of all persons affiliated with one organization (say UWisc) is almost identical to the set of all persons affiliated with another (UW-Madison), then it is likely these organizations are the same."* Such less precise rules cannot be expressed by traditional, precise languages like SQL. To support applications that use these less precise rules, a number of frameworks have been proposed that blend statistical assertions with traditional expressive languages, such as SQL or first-order logic, e.g., PRMs [14], BLOG [21], MLNs [29], PrDB [36], TUFFY [22]. These frameworks allow developers to write sophisticated programs that manipulate these less precisely understood sources of
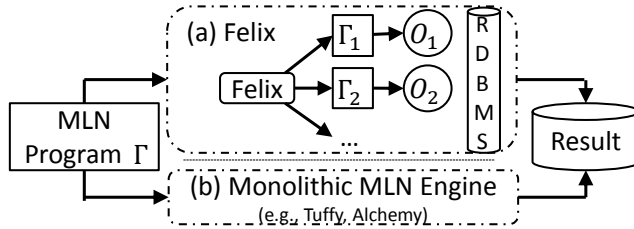
---

[1] http://www.wikipedia.com
[2] http://www.freebase.com

Figure 1: Felix Breaks an Input Program, Γ, into several, smaller programs (shown in Panel a), while prior approaches are monolithic (shown in Panel b).

data in a single language. We focus on one such framework, called *Markov Logic Networks*, that we have been using for almost two years as part of a DARPA five-year grand-challenge called *Machine Reading*, whose goal is to capture knowledge expressed in free-form text.

A compelling reason to use frameworks like Markov Logic Networks (MLNs) is that they have demonstrated high quality on semantically challenging tasks, e.g., Natural Language Processing [25, 39]. However, these languages are not a panacea as their performance and scalability have limited their use to small datasets. To develop the next generation of sophisticated applications, we argue that a promising approach is to improve the efficiency and scalability of such frameworks. Thus, we study techniques to scale up Markov Logic in the context of text applications.

The high-level idea of our approach to improve the scalability and performance of general-purpose statistical frameworks is *to exploit the existence a handful of common statistical subtasks*. We motivate this idea by example. A crucial subtask in text- or information-integration applications is *coreference resolution* (coref), in which we want to determine if two mentions refer to the same real-world entity, e.g., *"is Bill in one email the same as Bill Williams from accounting?"* Naïvely, coref of N entity mentions seems to require explicit representation of a quadratic search space that includes all $\binom{N}{2}$ pairs of mentions – which is untenable even for N in the tens of thousands. However, unaware that they are solving coref, current general-purpose frameworks do perform this quadratic processing and so cannot scale to large N. On the other hand, the problem of coref has been studied for decades, which has produced specialized algorithms that achieve both high quality and high performance [4, 6, 13]. This motivates the central technical question of this work: *"Can one combine the high quality and performance of specialized algorithms with the ease and flexibility of general-purpose frameworks?"*

Our system, FELIX, gives preliminary evidence that the answer is yes. Our first technical contribution is an architecture that allows us to view these statistical tasks as *operators*. In FELIX, each operator encapsulates an algorithm that consumes and produces relations. The inputs and outputs of the operators are tied together using standard SQL queries. Figure 1 indicates the difference at a high level between FELIX and prior approaches: monolithic approaches, such as ALCHEMY [29] or TUFFY [22], attack the entire input problem, while we decompose the problem into several small problems. Using specialized operators, FELIX executes complex IE programs on the DBLife dataset within several minutes with higher quality than DBLife's current rule-based approach (200% higher recall with the same precision) [12].

Immediate technical challenges to building FELIX include:

1. What is a good set of subtasks or operators?
2. How do we recognize such operators automatically given a program?
3. How do we execute those operators efficiently?

To answer those questions, our prototype of FELIX focuses on three specialized operators that are common in many text-processing applications: (1) *classification*, (2) *sequential labeling* (simply labeling), and (3) *coreference resolution*. We show that each of these operators is equivalent to a set of logical rules expressed in MLNs, and we describe our compiler that automatically decomposes an MLN into a directed acyclic graph (DAG) of such operators.

Decomposing sophisticated inference programs into smaller components is a classical idea, e.g., *Lagrangian Relaxation* dates back to the 60s [5, p. 244], *Belief Propagation* in the 1990s [43], and more recently *dual*

2

decomposition [33, 47]. FELIX builds on this work to derive algorithms to pass messages between the parts of the decomposed program. The novelty of our approach is that (similar to an RDBMS query optimizer) we propose to find such decompositions *automatically*, i.e., without the need for input from the user. Our second technical contribution is a set of sound compilation rules to find these operators. In general, the decision problems that a compiler must solve are properties of an infinite number of databases, and so it is not even clear that these properties are decidable. Nevertheless, for the analog of union of conjunctive queries, we establish that finding the above operators takes time at most exponential in the size of the input program.[3] We also develop sound (but not complete) heuristics to find these operators.

We observed an immediate data management challenge when we built FELIX. Even though each FELIX operator is efficient by itself, the scale of data passed between operators (via SQL queries) can be staggering: the reason is that statistical algorithms may produce huge numbers of combinations (say all pairs of potentially matching person mentions). FELIX uses an RDBMS which does increase scalability and performance using the RDBMS' sophisticated cost-based optimizer. Still, the sheer size of intermediate results are often killers for scalability, e.g., the complete input to coreference resolution on an Enron dataset has 1.2×10^{11} tuples.[4] The saving grace is that a downstream operator may only consume a small fraction of such intermediate data. In some cases we do not need to produce these massive, scalability-killing intermediate results. For example, a popular coref algorithm repeatedly asks *"given a fixed word x, tell me all words that are likely to be coreferent with x."* [3, 4]. Moreover, the algorithm only asks for a small fraction of the entire corpus. Thus, it would be wasteful to produce all possible matching pairs. Instead we can produce only those words that are needed on-demand (i.e., compute them lazily). On a dataset with 100K emails from Enron, we show that FELIX using a lazy approach finishes within 92 minutes, whereas an eager approach takes more than 10 hours and generates about 1 billion tuples. On the other hand, sometimes an eager strategy may be orders of magnitude more efficient. Thus, FELIX takes a cost-based approach to materialization.

FELIX considers a richer space of possible materialization strategies than simply eager or lazy: it can choose to eagerly materialize one or more subqueries responsible for data movement between operators [28]. Our third contribution shows that we can leverage the cost-estimation facility in the RDBMS coupled with our knowledge about FELIX's statistical operators to choose efficient query evaluation strategies in FELIX. On the DBLife and Enron datasets, our cost-based approach finds execution plans that achieve two orders of magnitude speedup over eager evaluation and 2-3X speedup compared to lazy evaluation.

To validate the efficacy of the operator-based approach, we perform an extensive experimental validation on three datasets that have been used by previous systems: (1) a webpage corpus from DBLife used by Cimple [12], (2) the Enron email datasets used by SYSTEMT [10], and (3) an NFL dataset used in the Machine Reading project. On all three datasets, FELIX achieves significantly higher quality (precision/recall) than the state-of-the-art approaches for a given task, e.g., 100% higher precision than SYSTEMT on the Enron email dataset with the same recall.[5] We take this as evidence that the operator-based approach is a promising step toward scaling up statistical-based approaches.

2 Preliminaries

To illustrate how MLNs can be used in text-processing applications, we describe a program that extracts affiliations from the Web. We then describe how specialized subtasks arise in MLN programs, including classification, labeling, and coreference resolution.

---

[3]The underlying decision problems are in $\Pi_2 \mathbf{P}$, and we establish that some are complete for $\Pi_2 \mathbf{P}$.

[4]http://bailando.sims.berkeley.edu/enron_email.html

[5]FELIX is available for download to replicate these experiments from http://www.cs.wisc.edu/hazy/felix/. This site also contains a full version of the paper with proofs and extended experimental results.

Schema	Evidence	Rules

		weight	**rule**	
pSimHard(per1, per2)	coOccurs('Ullman', 'Stanford Univ.')	$+\infty$	$\texttt{pCoref}(p,p)$	(F_1)
pSimSoft(per1, per2)	coOccurs('Jeff Ullman', 'Stanford')	$+\infty$	$\texttt{pCoref}(p1,p2) => \texttt{pCoref}(p2,p1)$	(F_2)
oSimHard(org1, org2)	coOccurs('Gray', 'San Jose Lab')	$+\infty$	$\texttt{pCoref}(x,y), \texttt{pCoref}(y,z) => \texttt{pCoref}(x,z)$	(F_3)
pSimSoft(org1, org2)	coOccurs('J. Gray', 'IBM San Jose')	6	$\texttt{pSimHard}(p1,p2) => \texttt{pCoref}(p1,p2)$	(F_4)
coOccurs(per, org)	coOccurs('David', 'UW-Madison')	2	$\texttt{affil}(p1,o), \texttt{affil}(p2,o), \texttt{pSimSoft}(p1,p2) => \texttt{pCoref}(p1,p2)$	(F_5)
homepage(per, page)	coOccurs('David', 'UWisc')			
oMention(page, org)	coOccurs('Jeff', 'UWisc')	$+\infty$	$\texttt{faculty}(o,p) => \texttt{affil}(p,o)$	(F_6)
faculty(org, per)	faculty('MIT', 'Chomsky')	8	$\texttt{homepage}(p,d), \texttt{oMention}(d,o) => \texttt{affil}(p,o)$	(F_7)
*affil(per, org)	homepage('Joe', 'Doc201')	3	$\texttt{coOccurs}(p,o1), \texttt{oCoref}(o1,o2) => \texttt{affil}(p,o2)$	(F_8)
*oCoref(org1, org2)	oMention('Doc201', 'IBM')	4	$\texttt{coOccurs}(p1,o), \texttt{pCoref}(p1,p2) => \texttt{affil}(p2,o)$	(F_9)
*pCoref(per1, per2)	$\cdots$	$\cdots$		

Figure 2: An example Markov Logic program that performs three tasks jointly: 1. discover affiliation relationships between people and organizations (`affil`); 2. resolve coreference among people mentions (`pCoref`); and 3. resolve coreference among organization mentions (`oCoref`). The remaining eight relations are evidence relations. In particular, `coOccurs` stores raw extraction results of person-organization co-occurrences.

2.1 Markov Logic Networks in Felix

Consider the task of extracting affiliation relationships between people and organizations from Web text. Beginning with raw text, an IE system, such as Cimple [12] or SYSTEMT [10], first attempts to extract all person and organization mentions. Transforming the raw text into clean relations is difficult. For example, a major challenge is that a single real-world entity may be referred to in many different ways, e.g., *"UWisc"* and *"University of Wisconsin"* refer to the same real-world entity. Once we have associated a mention to an entity, we can perform more sophisticated reasoning, e.g., it is likely that a person is affiliated with only a small number of organizations. Below, we describe how MLNs can be used to simultaneously perform both relationship discovery and coreference resolution that allow us to improve the quality of an IE system. Moreover, these MLNs allow us to perform more sophisticated inference than a typical IE system, which we have found useful in applications.

Our system FELIX is a middleware system: it takes as input a standard MLN program, performs statistical inference, and outputs its results into one or more relations that are stored in a PostgreSQL database. An MLN program consists of three parts: schema, evidence, and rules. To tell FELIX what data will be provided or generated, the user provides a *schema*. Some relations are standard database relations, and we call these relations *evidence*. Intuitively, evidence relations contain tuples that we assume are correct. In the schema of Figure 2, the first eight relations are evidence relations. For example, we know that 'Ullman' and 'Stanford Univ.' co-occur in some webpage, and that 'Doc201' is the homepage of 'Joe'. Other evidence includes string similarity information. In addition to evidence relations, there are also relations whose content we do not know, but we want the MLN program to predict; they are called *query relations*. In Figure 2, `affil` is a query relation since we want the MLN to predict affiliation relationships between persons and organizations. The other two query relations are `pCoref` and `oCoref`, for person and organization coreference, respectively.

In addition to the schema and evidence, we also provide a set of MLN rules to encode our knowledge about the correlations and constraints over the relations. An MLN rule is a first-order logic formula associated with an extended-real-valued number called a *weight*. Infinite-weighted rules are called hard rules, which means that they must hold in any prediction that the MLN engine makes. In contrast, rules with finite weights are soft rules: a positive weight indicates confidence in the rule's prediction.[6]

Example 1 An important type of hard rule is a standard SQL query, e.g., to transform the results for use in the application. A more sophisticated examples of hard rule is to encode that coreference is a transitive property, which is captured by the hard rule F_3. Rules F_8 and F_9 use person-organization co-occurrences (`coOccurs`)

---

[6]Roughly these weights correspond to the log odds of the probability that the statement is true. (The log odds of p is $\log \frac{p}{1-p}$.) In general, these weights do not have a simple probabilistic interpretation [29]. In FELIX, weights can be set by the user or automatically learned. We do not discuss learning in this work.

together with coreference (`pCoref` and `oCoref`) to deduce affiliation relationships (`affil`). These rules are soft since co-occurrence in a webpage does not necessarily imply affiliation.

Intuitively, when a soft rule is violated, we pay a *cost* equal to its weight (described below). For example, if `coOccurs`('Ullman', 'Stanford Univ.') and `pCoref`('Ullman', 'Jeff Ullman'), but not `affil`('Jeff Ullman', 'Stanford Univ.'), then we pay a cost of 4 because of F_9. An MLN inference algorithm's goal is to find a prediction that minimizes the sum of such costs.

Similarly, affiliation relationships can be used to deduce non-obvious coreferences. For instance, using the fact that 'David' is affiliated with both 'UW-Madison' and 'UWisc', FELIX may infer that 'UW-Madison' and 'UWisc' refer to the same organization (rules on `oCoref` are omitted from Figure 2). If FELIX knows that 'Jeff' co-occurs with 'UWisc', then it is able to infer 'Jeff''s affiliation with 'UW-Madison'.

Semantics An MLN program defines a probability distribution over database instances (possible worlds). Formally, we first fix a schema σ (as in Figure 2) and a domain D. Given as input a set of formula $\bar{F} = F_1, \ldots, F_N$ with weights $w_1, \ldots, w_N$, they define a probability distribution over *possible worlds* (deterministic databases) as follows. Given a formula F_k with free variables $\bar{x} = (x_1, \cdots, x_m)$, then for each $\bar{d} \in D^m$, we create a new formula $g_{\bar{d}}$ called a *ground formula* where $g_{\bar{d}}$ denotes the result of substituting each variable x_i of F_k with d_i. We assign the weight w_k to $g_{\bar{d}}$. Denote by $G = (\bar{g}, w)$ the set of all ground formulae of $\bar{F}$. We call the set of all tuples in G the *ground database*. Let w be a function that maps each ground formula to its assigned weight. Fix an MLN $\bar{F}$, then for any possible world (instance) I we say a ground formula g is *violated* if $w(g) > 0$ and g is false in I, or if $w(g) < 0$ and g is true in I. We denote the set of ground formulae violated in a world I as $V(I)$. The cost of the world I is

$$\text{cost}_{mln}(I) = \sum_{g \in V(I)} |w(g)| \tag{1}$$

Through cost_{mln}, an MLN defines a probability distribution over all instances using the exponential family of distributions (that are the basis for graphical models [43]):

$$\Pr[I] = Z^{-1} \exp \{-\text{cost}_{mln}(I)\}$$

where Z is a normalizing constant.

Inference There are two main types of inference with MLNs: *MAP (maximum a posterior) inference*, where we want to find a most likely world, i.e., a world with the lowest cost, and *marginal inference*, where we want to compute the marginal probability of each unknown tuple. In our previous work [22], we use an RDBMS to build a scalable MLN inference engine, TUFFY, that supports both MAP and marginal inference. TUFFY is an operator in FELIX, and so FELIX can perform both types of inference. In this work, we describe MAP inference to simplify the presentation.

2.2 Specialized Subtasks

We describe some common patterns that we have seen in our use of Markov Logic over the past two years. Not coincidentally, these patterns form the logical operators in FELIX. As an example, we consider a simplified application of the Machine Reading project: *extract the winner and loser of an NFL game on a certain day from news articles about football.* We describe one implementation of this task:

Labeling A common subtask in text application is to label a sequence of tokens in a document. Here, we label each phrase in the document with winner (W), loser (L), or other (O). A simplified program is the following:

$$\begin{array}{lll} \infty & \texttt{label}(d, p, l1), \texttt{label}(d, p, l2) => l1 = l2 & (\gamma_{2.2.1}) \\ 10 & \texttt{next}(d, p, p'), \texttt{token}(p', \text{'win'}) => \texttt{label}(d, p, W) & (\gamma_{2.2.2}) \\ 10 & \texttt{next}(d, p, p'), \texttt{token}(p', \text{'loss'}) => \texttt{label}(d, p, L) & (\gamma_{2.2.3}) \\ 1 & \texttt{label}(d, p1, W), \texttt{next}(d, p1, p2) =>!\texttt{label}(d, p2, W) & (\gamma_{2.2.4}) \end{array}$$
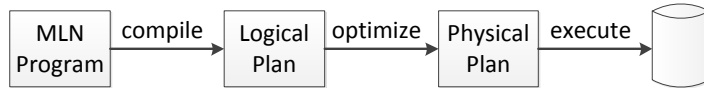
Figure 3: Execution Pipeline of Felix

The first rule ($\gamma_{2.2.1}$) indicates that every phrase (p) in every document (d) should be labeled with at most a single label l (a key constraint in every possible world). The second rule says that if one phrase is followed by a token *'win'*, it is more likely to be labeled as a winner (W). (Here $\mathtt{next}(\mathtt{d}, \mathtt{p}, \mathtt{p}')$ means that phrase p' is the immediate successor of p' in document d). The fourth rule says that if a phrase is labeled W, it is less likely for the next phrase to be also labeled W.

We show in the full version of this article that these rules define an instance of the same inference problem as a Conditional Random Field (CRF) [17]. This is a significant win as there are efficient dynamic programming-based algorithms that can solve both MAP and marginal inference of CRFs [17]. FELIX implements these algorithms.

Classification Another subtask in text applications is classification. The example here is to classify each team as a winner ($\mathtt{winner}$) of a fixed game (we omit the logic specifying the game for clarity). A program may use the following rules for this subtask:

$$10 \quad \mathtt{label}(p, W), \mathtt{referTo}(p, team) => \mathtt{winner}(team)$$
$$10 \quad \mathtt{label}(p, L), \mathtt{referTo}(p, team) => \mathtt{!winner}(team)$$

where $\mathtt{label}(p, l)$ is the result of labeling in the previous example, $\mathtt{referTo}(phrase, team)$ maps each phrase to the team entity it may refer to, and $\mathtt{winner}(t)$ says that team t was the winner of a fixed game.

These rules define a classifier, which can then be implemented using efficient physical implementations, e.g., a logistic regressor. Thus, we could compute the exact probability of $\mathtt{winner}(team)$ for each team *team* using simple SQL aggregates (since inference for logistic regression is simply a weighted sum of features followed by a comparison). On the other hand, unaware of this subtask, a monolithic MLN system would run sample-based inference algorithms that produce only approximate answers.

Coreference Resolution A third common subtask is coreference resolution, e.g., given a set of strings (say phrases in a document) we want to decide which strings represent the same real-world entities (say team). These tasks are ubiquitous in text processing. Consider the following rules:

$$\infty \quad \mathtt{coRef}(p1, p2), \mathtt{coRef}(p2, p3) => \mathtt{coRef}(p1, p3)$$
$$\infty \quad \mathtt{coRef}(p1, p2), => \mathtt{coRef}(p2, p1)$$
$$\infty \quad \mathtt{coRef}(p1, p1)$$
$$5 \quad \mathtt{inSameDoc}(p1, p2), \mathtt{subString}(p1, p2) => \mathtt{coRef}(p1, p2)$$

where $\mathtt{inSameDoc}(p1, p2)$ means $p1$ and $p2$ appear in a same document, $\mathtt{subString}(p1, p2)$ means $p1$ has $p2$ as a sub-string, and $\mathtt{coRef}(p1, p2)$ is the coreference relation. The first three rules declare that the coreference relation is transitive, symmetric, and reflexive. The fourth rule says that phrases in the same document tend to refer to the same entity if one string is a sub-string of the other (e.g., *'Green Bay'* and *'Green Bay Packer'*). A real coreference application would likely have many such rules of varying weight.

Coreference resolution is a well-studied problem [4, 13, 26]. The underlying inference problem is NP-hard in almost all variants. As a result, there is a literature on approximation techniques. FELIX recognizes this subtask and executes these rules with an approximation technique (e.g., *correlation clustering* [3, 4]) instead of a generic MLN inference algorithm.

Operator	Implementation
Classification	Logistic Regression [7]
Coreference	Correlation Clustering [3, 4]
Labeling	Conditional Random Field [17]
Generic MLN inference	TUFFY [22]

Table 1: FELIX operators and an example implementation for each operator.

3 Architecture of Felix

We describe the architecture of FELIX whose goal is to execute an MLN program using a pre-defined handful of operators (see Figure 1). To execute an MLN program, FELIX transforms the input program Γ in several phases as illustrated in Figure 3. Each of the phases essentially mirrors the textbook phases in an RDBMS: FELIX first *compiles* a Markov Logic program into a *logical plan of statistical operators*. Then, FELIX performs *optimization* (code selection). The output of code selection is a sequence of statements that are then executed (by the *executor*). In turn, the executor may call an RDBMS (PostgreSQL in our case) or special-purpose operator code.

In the remainder of this section, we describe key concepts in FELIX (Section 3.1), namely, statistical operators and their implementations. Then in Section 3.2, we walkthrough how FELIX executes an MLN program.

3.1 Statistical Operators

A key concept in FELIX is a *Statistical Operator* that encapsulates a single statistical task, e.g., coreference resolution, classification, or generic MLN inference (Table 1). Statistical operators are similar to operators in an RDBMS: they take as input one or more relations and produce another relation. Then in a second phase, operators are instantiated with particular pieces of code; we call the operator together with its implementation an *implementation*. Similar to a standard RDBMS, the process of selecting an implementation for an operator is called *code selection*. After code selection in FELIX, the algorithmic details of a statistical operator remain hidden to the FELIX optimizer, but an an operator does expose its data access patterns via *adorned views* (described below).

Logical-Statistical Operators A *logical-statistical operator* takes as input one or more relations and produces as output one or more relations. The input relations are specified by Datalog-like queries $Q_1(\bar{x}_1), \ldots, Q_N(\bar{x}_N)$. The body of a query Q_i may refer either to *evidence relations*, which are standard database tables, or to the output of other statistical operators. We refer to each Q_i as a *data movement operator* to avoid overloading the term query. As shown in Table 1, FELIX considers three specialized operators, *Classification*, *Labeling*, and *Coreference Resolution*, and one generic, catch-all operator that performs generic MLN inference. We define each of the specialized operators below and illustrate the difference between each statistical operator and its namesake statistical task.

The standard statistical task of **Classification** can be formalized as follows: A *Boolean Classifier* is a pair (X, y) where X is a set of random variables (called *covariates*) and y is a Boolean random variable (the *outcome*) such that $\Pr[y = 0|X] + \Pr[y = 1|X] = 1$. In FELIX, a classification operator has an input relation I and an output relation O; it satisfies the property that in any ground database each tuple of O corresponds to an independent classification task.[7]

The task of **Labeling** is a generalization of classification where there are multiple (correlated) outcomes. Formally, a labeling is defined by a triple $(X, Y, \sqsubseteq)$ where X is a set of random variables (covariates), Y is a set of random variables (outcomes), and $\sqsubseteq$ is an order on Y that specifies the correlations of the outcomes. If $\sqsubseteq$ is

---

[7]In FELIX, a classification may be multiclass as well.

a total order, this means $y_1, \ldots, y_N$ is such that $y_i \sqsubseteq y_j$ implies $i \leq j$ then

$$\Pr[Y_{[1,N]}|y_i, X] = \Pr[Y_{[1,i-1]}|y_i, X] \Pr[Y_{[i+1,N]}|y_i, X]$$

where $Y_{[i,j]} = \{y_i, y_{i+1}, \ldots, y_j\}$. This is a conditional independence criterion ($Y_{[1,i-1]}$ and $Y_{[i+1,N]}$ are independent conditioned on y_i). This condition is easy to generalize to the case when $\sqsubseteq$ is a partial order [11]. In FELIX, a *labeling* operator with an output relation O implies that in the ground database D that corresponds to the input program Γ the relation O can be partitioned into one or more independent labeling tasks.

The task of **Coreference Resolution** can be defined as follows: given a pair of sets (X, Y) partition Y conditioned on X (or produce a distribution over partitions). The idea is that if y_i and y_j are in the same partition then y_i and y_j refer to the same entity (e.g., are the same person or refer to the same organization, etc.) A Coreference Resolution operator insists that its output relation can be decomposed into one or more coreference resolution problems when grounded for any input database.

The final operator in FELIX is a a **Generic MLN inference** operator that can take as input any MLN.

The connection between an operator Θ and the corresponding standard statistical task T is that the operator Θ requires that for any input database the inference problem for each tuples in the output relation of Θ can be decomposed in to a set of independent problems of type T. Given an MLN Γ, deciding when such a decomposition holds is non-trivial; it is the heart of FELIX's compiler and is the subject of Section 4.1.

Physical-Statistical Operators (Implementations) During code selection FELIX chooses an implementation for each operator. Table 1 shows an example implementation for each FELIX operator. A long-term design goal for FELIX is to support many implementations for each logical operator (for clarity in this work, we discuss only a single implementation). To facilitate this goal, we need to hide the details of the underlying implementation. Of course, there is a tradeoff: we can expose more information which enables more optimization opportunities, but more information makes it difficult to add new algorithms. Currently, FELIX takes a minimalist approach and only exposes the data access patterns of each implementation.

To expose the data access patterns of an operator, FELIX uses *adorned views* [42]. In an adorned view, each variable in the head of a view definition is associated with a binding-type, which is either b (bound) or f (free). Given a query Q_i, denote by $\bar{x}^{\mathsf{b}}$ (resp. $\bar{x}^{\mathsf{f}}$) the set of bound (resp. free) variables in its head. Then we can view Q_i as a function mapping an assignment to $\bar{x}^{\mathsf{b}}$ (i.e., a tuple) to a set of assignments to $\bar{x}^{\mathsf{f}}$ (i.e., a relation). To concisely denote binding patterns, we follow Ullman [42]. A query Q of arity $a(Q)$ is written as $Q^{\alpha}(\bar{x})$ where $\alpha \in \{\mathsf{b}, \mathsf{f}\}^{a(Q)}$. The binding patterns are a property of the implementation that we select in code-selection – not the logical operator.

Example 1 A Coreference operator initially may have a data-movement operator similar to F_5 in Figure 2 that says if two persons (x and y) are affiliated with the same organization (o), then there is some chance they are the same person. Thus, the operator takes as input a data-movement operator as the following:

$$\texttt{SamePerson}(x, y) \quad \leftarrow \quad \texttt{affil}(x, o), \texttt{affil}(y, o).$$

Say that FELIX chooses to instantiate this operator with correlation clustering [3,4]. At this point, it knows the data access properties of that algorithm (which essentially ask only for "neighboring" elements). FELIX represents this using the following annotated data movement operator:

$$\texttt{SamePerson}^{\mathsf{bf}}(x, y) \quad \leftarrow \quad \texttt{affil}(x, o), \texttt{affil}(y, o).$$

which is adorned as bf. Thus, during execution, the implementation of the Coref operator will send to this data-movement operator requests such as $x = $'Jeff', and expect to receive a set of names $\{y | \mathsf{Q}(\text{'Jeff'}, y)\}$ that share affiliations with 'Jeff'. FELIX uses this information to better optimize the data access properties (see Section 4.2).
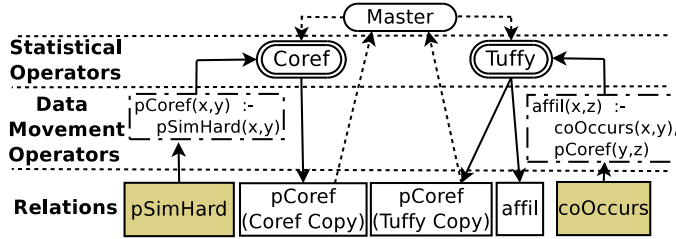
Figure 4: An example logical plan. Relations in shaded boxes are evidence relations.

3.2 Felix Walkthrough

Recall that the input to FELIX is an MLN program Γ, and the output of FELIX is an instantiation of the query relations specified in Γ. There are three phases that FELIX goes through to instantiate these relations: (1) compilation, (2) optimization, and (3) execution. We describe them in the order that FELIX performs them.

(1) Compilation takes as input an MLN program and returns a logical plan that is essentially a graph whose nodes are *relations*, *data-movement operators*, and *statistical operators*. A relation may be either an evidence relation or a relation output by a statistical operator. One data-movement operator Q_i is created for each input in a statistical operator (thus, the same query may be replicated multiple times); the node representing Q_i has an edge to the node that represents its corresponding statistical operator. There is an edge from each base relation mentioned in the body of Q_i to Q_i. For each query predicate that is shared by two operators, FELIX creates a copy. (The copy is used during execution, see Section 4.3). Figure 4 illustrates an example logical plan where `pCoref` is shared. We ensure that a logical plan is a DAG. This DAG is output by compilation and then consumed by the next phase, *optimization*. Compilation is described in Section 4.1.

(2) Optimization takes as input a DAG of operators and produces a DAG of *statements*. Statements are of two forms: (1) a prepared SQL statement or (2) a statement encoding the necessary information to run a statistical operator; e.g., the number of iterations that a TUFFY operator should run, where it should fetch its data from, etc. A key responsibility of *optimization* is to decide on whether or not to materialize intermediate results of statistical operators (and which portions of these results to materialize). It is in this phase that *implementations* (physical operators) are used. This is the subject of Section 4.2. From a single data-movement operator Q, the Optimization phase may produce several different adornments Q^α. For each adorned data-movement operator Q^α, it may produce several statements (e.g., if FELIX materializes several subqueries of Q^α).

(3) Execution takes the DAG of statements produced by the optimization phase and executes them using PostgreSQL (for SQL statements), TUFFY (for MLN statements), or the code for a particular operator. FELIX supports two types of parallelization: 1) Operator-level parallelism: If two operators in the DAG are independent of one another, FELIX can run them in parallel; and 2) Data-level parallelism: each operator itself may partition the data and run sub-operators in parallel internally. A key responsibility of execution is to handle the communication between the operators. We refer to the process that manages these activities as the *Master*. The role of the Master is described in Section 4.3.

4 The Stages of Felix

We describe our technical contributions in the three phases of FELIX: *compilation*, *optimization*, and *execution*.

9

Annotation	Symbol	Example
Reflexive	REF	$p(x,y) \implies p(x,x)$
Symmetric	SYM	$p(x,y) \implies p(y,x)$
Transitive	TRN	$p(x,y), p(y,z) \implies p(x,z)$
Key	KEY	$p(x,y), p(x,z) \implies y = z$
Not Recursive	NoREC	Can be defined w/o Recursion.
Tree Recursive	TrRec	See Equation 2

Table 2: Annotations assigned to predicates by the Felix compiler. TrRec is a derived annotation. The definitions should be read with universal quantifiers in front of x, y, z. KEY refers to a non-trivial key.

Operator Type	Required Properties
CoRef	REF, SYM, TRN
Classification	KEY, NoREC
Labeling	KEY, TrREC
Generic MLN Inference	none

Table 3: Operators and their required properties.

4.1 Compilation and Logical Optimization

The compiler of FELIX takes as input an MLN program Γ and produces a graph where the nodes are *statistical operators* $O_1, \ldots, O_m$ and an edge indicates a data flow dependency. This graph is logically equivalent to the input program. Intuitively, each operator in the graph is responsible for inference on a subset of predicates in Γ.

FELIX's compiler is responsible for a host of tasks: it determines the data-movement operators that are input to each operator O_i, it performs the static analysis necessary to exploit data-level parallelism, etc. But, the most interesting job is how the compiler discovers properties of predicates in the input Markov Logic program, Γ, that enable FELIX to decompose Γ into operators.

To decompose Γ into operators, FELIX uses a two-step approach. FELIX's first step is to annotate each predicate p with a set of *properties*. An example property is whether or not p is symmetric. Table 2 lists of the set of properties that FELIX attempts to discover with their definitions. This is non-trivial as the predicates are the output of SQL queries (or formally, datalog programs). Once the properties are found, FELIX uses Table 3 to list all possible options for an operator.[8] When there are two options, FELIX chooses the first operator to appear in the following order (Coref,Classification,Labeling,TUFFY). This order intuitively favors more specific operators.

Detecting Properties The most technically difficult part of the compiler is determining the properties of the predicates. There are two types of properties that FELIX looks for: (1) schema-like properties of any possible worlds that satisfies Γ and (2) schema-like properties of the resulting ground database. For both types of properties, the challenge is that we must infer these properties from the underlying rules applied to an infinite set of databases.[9] For example, SYM is the property:

"*for any database I that satisfies Γ, does the sentence $\forall x, y.\mathrm{pCoref}(x,y) \iff \mathrm{pCoref}(y,x)$ hold?*".

Since I comes from an infinite set, it is not immediately clear that the property is even decidable. Indeed, some properties in Table 2 are not decidable for Markov Logic programs.

Although the set of properties in Table 2 is motivated by considerations from statistical inference, the properties depend *only on the hard rules in* Γ, i.e., the constraints and (SQL-like) data transformations in the

---

[8]The proofs of correctness of these rules are straightforward and are in the full version of this paper.

[9]As is standard in database theory [2], to model the fact the query compiler runs without examining the data, we consider the domain of the attributes to be unbounded. If the domain of each attribute is known then, all of the above properties are decidable by the trivial algorithm that enumerates all (finitely many) instances.

program. Let Γ_∞ be the set of rules in Γ that have infinite weight. We consider the case when Γ_∞ is written as a datalog program.

Theorem 4.1. *Given a datalog program Γ_∞, a predicate p, and a property θ deciding if for all input databases p has property θ is undecidable if $\theta \in \{REF, SYM\}$.*

The above result is not surprising as datalog is a powerful language and containment is undecidable [2, ch. 12] (the proof reduces from containment). Moreover, the compiler are related to *implication problems* studied by Abiteboul and Hull (who also establish that generalizations of what we call the KEY and TRN problem are undecidable [1]). NoRec is the negation of the *boundedness problem* [9] which is undecidable.

In many cases, recursion is not used in Γ_∞ (e.g., Γ_∞ may contain of standard SQL queries that transform the data), and so a natural restriction is to consider Γ_∞ without recursion, i.e., as a union of conjunctive queries.

Theorem 4.2. *Given a union of conjunctive queries Γ_∞, deciding if for all input databases that satisfy Γ_∞ the query predicate p has property θ where $\theta \in \{REF, SYM\}$ (Table 2) is decidable. Furthermore, the problem is $\Pi_2 P$-Complete. KEY and TRN are trivially false. NoRec is trivially true.*

Still, FELIX must produce annotations. To cope with the undecidability and intractability of finding our compiler annotations, FELIX uses a set of sound (but not complete) rules that are described by simple patterns. For example, we can conclude that a predicate R is transitive if the rules contain syntactically the rule $R(x, y), R(y, z) => R(x, z)$ with weight ∞. A complete list of such expressions is in the full version of this paper.

Ground Structure The second type of properties that FELIX considers characterize the graphical structure of the *ground database* (in turn, this structure describes the correlations that must be accounted for in the inference process). We assume that Γ is written as datalog program (with stratified negation). The ground database is a function of both soft and hard rules in the input program, and so we consider both types of rules here. FELIX's compiler attempts to deduce a special case of recursion that is motivated by (tree-structured) conditional random fields that we call TrREC. Suppose that there is a single recursive rule that contains p in the body and the head is of the form:

$$p(x, y), T(\underline{y}, z) => p(x, z) \tag{2}$$

where the first attribute of T is a key and the transitive closure of T is a partial order. In the ground database, p will be "tree-structured". MAP and marginal inference for such rules are in P-time [36, 43]. FELIX has a regular expression to deduce this rule.

We illustrate the compilation process by example.

Example 1 Consider the labeling example in Section 2.2, the relation `label`($phrase, label$) is labeled as KEY (from ($\gamma_{2.1.1}$)) and we get TrREC from deducing that (d, p) and (d, p') are both candidate keys of `next`(d, p, p') (and there are no other syntactically recursive rules for `next`). So, according to Table 3, `label` can be solved by the Labeling operator or the generic MLN inference algorithm. FELIX chooses the Labeling operator as it is more specific than TUFFY.

4.2 Physical Optimization: Cost-based Materialization

The goal of physical optimization is to choose an implementation for each operator in the logical plan that is output by the compiler. In full analogy with a traditional RDBMS, after selecting an implementation for each operator we call the plan a *physical plan*. In FELIX, data are passed between statistical operators using data-movement operators which are SQL queries. Once we have created an initial physical plan, FELIX knows each operator's binding patterns, and so in this phase we can produce data-movement operators. Since the data-movement operators are SQL queries, we get query optimization for each operator from the RDBMS for

free. Still, we have observed that a critical bottleneck in FELIX's execution is the efficiency of individual data-movement operators. One critical optimization that we describe in this section is that FELIX may choose to materialize (or not) some portion of these SQL queries.

Recall that RDBMSs can execute queries both eagerly (using standard SQL) and lazily (via prepared statements). We have found that both types of execution are helpful when executing FELIX plans. If an operator needs repeated access to the entire result of Q_i, it is often more efficient to let the RDBMS produce all of the results and materialize them in an intermediate relation. On the other hand, some statistical operators may inspect only a small fraction of their search space and so such eager materialization is inefficient. For example, one implementation of the Coref operator is a stochastic algorithm that examines roughly linear in the number of nodes (even though the input to Coref contains a quadratic number of pairs of nodes) [4]. In some cases, this input may be so large that an eager materialization strategy would exhaust the available disk space. For example, on an Enron dataset, materializing the following query would require over 1TB of disk space:

$$\texttt{maylink}^{\texttt{bb}}(x,y) \quad \leftarrow \quad \texttt{mention}(x, name1), \texttt{mention}(y, name2),$$
$$\texttt{mayref}(name1, z), \texttt{mayref}(name2, z).$$

FELIX is, however, not confined to fully eager or fully lazy. In FELIX, we have found that intermediate points (e.g., materializing a subquery of Q_i) can have dramatic speed improvements (see Section 6.3). We describe FELIX's cost-based optimizer that explores a search space of intermediate results; similar to a System-R-style cost-based RBDMS optimizer, FELIX enumerates the plan space and chooses the plan with the lowest (predicted) cost. Almost all cost estimation for a given SQL query is done by the underlying RDBMS (here, PostgreSQL), while FELIX exploits these estimates from PostgreSQL to search for the optimal plan.

Felix Cost Model   To define our cost model, we introduce some notation. Let $Q(\bar{x}) \leftarrow g_1, g_2, \ldots, g_k$ be a data-movement operator. Let $G = \{g_i | 1 \leq i \leq k\}$ be the set of subgoals of Q. Let $\mathcal{G} = \{G_1, \ldots, G_m\}$ be a partition of G; i.e., $G_j \subseteq G$, $G_i \cap G_j = \emptyset$ for all $i \neq j$, and $\bigcup G_j = G$. Intuitively, a partition represents a possible materialization strategy: each element of the partition represents a query (or simply a relation) that FELIX is considering materializing. That is, the case of one $G_i = G$ corresponds to a fully eager strategy. The case where all G_i are singleton sets corresponds to a lazy strategy.

More precisely, define $Q_j(\bar{x}_j) \leftarrow G_j$ where $\bar{x}_j$ is the set of variables in G_j shared with $\bar{x}$ or any other G_i for $i \neq j$. Then, let query $Q'(\bar{x}) \leftarrow Q_1, \ldots, Q_m$. Let t be the total number of lazy incremental evaluations of Q' performed by the statistical operator. Then, we model Q' contribution to the execution cost as:

$$\text{ExecCost}(Q', t) = t \cdot \text{Inc}_Q(Q') + \sum_{i=1}^{m} \text{Mat}(Q_i)$$

$\text{Mat}(Q_i)$ is the cost of eagerly materializing Q_i and $\text{Inc}_Q(Q')$ is the average cost of lazy incremental evaluation on Q'.

Both t and $\text{Inc}_Q(Q')$ depend on the access pattern of the statistical operator that uses Q. To optimize Q in FELIX, we must be able to estimate both parameters. In principle, both of these parameters can be estimated by the RDBMS optimizer. A significant implementation detail is that since the subgoals in Q' are not actually materialized, we cannot directly ask PostgreSQL for the incremental cost $\text{Inc}_Q(Q')$.[10] In our prototype version of FELIX, we implement a simple approximation of PostgreSQL's optimizer (that assumes incremental plans use only index-nested-loop joins), and so our results should be taken as a lower bound on the performance gains that are possible using a cost-based approach.

With this cost model, the second half of our optimizer is to enumerate different plans. The number of possible plans is exponential in the size of the largest rule in an input Markov Logic program, but in our applications the individual rules are small. Thus, we can estimate the cost of each alternative, and we pick the one with lowest ExecCost.

---

[10]PostgreSQL does not fully support "what-if" queries, although other RDBMS do, e.g., for indexing tuning.

4.3   Execution

Given a physical plan, there are two phases in execution: (1) *initialization*, where we initialize each operator with an initial estimate of each query relation, and (2) *iteration*, where we perform inference and pass messages. During iteration, all operators can execute in parallel.

Initialization   Similar to how an RDBMS executes a physical plan of relational operators, statistical operators can run in parallel (subject to data flow dependencies). FELIX starts from the statistical operators at the leaves of the plan (which do not depend on others statistical relations). FELIX then traverses the DAG in a breadth-first order to greedily populate all relations in the plan.

Iteration   For iteration, FELIX's execution strategy is based on the classical technique of *decomposition* from the mathematical programming literature [5, 47]. To use this literature, we formally represent MLN inference as a mathematical optimization program.[11]   Given a desired decomposition, dual decomposition generates a program with the same optimal (with possibly redundant constraints). This type of decomposition has recently been recognized as the unifying formal underpinnings for many variants of this idea, e.g., the *belief propagation* family of algorithms from graphical models [43, p. 86]. Given a decomposition, there are many execution strategies, in this work we explore a simple strategy for MAP inference called *Lagrangian Relaxation*.
   We describe decomposition of MAP inference by example and defer marginal inference to the full paper.

Example 2 Consider a simple Markov Logic program:

$$\{(R() : w_r), (S() : w_s), (T() : w_t)\}$$

subject to the hard constraint that $R() \iff \neg S()$ and $S() \iff \neg T$. We introduce three Boolean variables r, s, t. Then MAP inference is equivalent to the following Boolean integer program:

$$\max_{r,s,t} w_r r + w_s s + w_t t \text{ s.t. } r \neq s, t \neq s$$

Let us decompose this problem. Let $f_1(r, s)$ be the function that is negative infinity if $r = s$ otherwise is $w_r r + w_s s/2$. Similarly for f_2. Then our problem is equivalent to:

$$\max_{r,s,t} f_1(r, s) + f_2(s, t)$$

We think of each f_i as an operator that is trying to solve its own local minimization problem. The challenge is that the variable s is shared by both functions. Decomposition suggests that we create two copies of the variable s (s_1 and s_2) and consider the equivalent program

$$\max_{r,s_1,s_2,t} f_1(r, s_1) + f_2(s_2, t) \text{ s.t. } s_1 = s_2$$

While this program is equivalent, it is in the syntactic form to relax the constraints. One method is *Lagrangian relaxation* that introduces a scalar λ called a Lagrange multiplier for each such constraint. Define $g(\lambda)$ as

$$g(\lambda) = \max_{r,s_1,s_2,t} f_1(r, s_1) + f_2(r_2, t) + \lambda(s_1 - s_2)$$

Then, $\min_{\lambda \geq 0} g(\lambda)$ is our dual optimization problem. If the resulting solution is feasible for the original program (i.e., satisfies the constraints), then the optimal of this dual constraint will be the same as the primal optimal [47, p. 168].

---

[11]This mathematical program is never constructed. We use it as a formal device to devise a message-passing protocol.

For any fixed λ, this technique decomposes the problem into two problems (each problem is grouped by parentheses):

$$g(\lambda) = \left(\max_{r,s_1} f_1(r, s_1) + \lambda s_1 \right) + \left(\max_{s_2,t} f_2(s_2, t) - \lambda s_2 \right)$$

We can now solve $g(\lambda)$ using whatever optimization method we choose, e.g., gradient descent [47, p. 174]. At each iteration, this amounts to simply updating the λ that corresponds to each pairwise equality constraint. These λ form the messages that are passed between operators.

The final observation is that to solve the individual f_is we can use existing MLN inference techniques. In this example, our two programs Γ_1 and Γ_2 become:

$$\Gamma_1 = \{(R() : w_r), (S_1() : (w_s/2 + \lambda)), R() \iff \neg S_1()\} \qquad \text{and}$$
$$\Gamma_2 = \{(T() : w_t), (S_2() : (w_s/2 - \lambda)), S_2() \iff \neg T()\}$$

One can check that these Markov Logic programs have the same optimal as the above integer programs. This fact is critical: it implies that we can reuse MLN inference algorithms on the subproblems.

FELIX's algorithm is a straightforward adaptation of the above example. For simplicity, assume that each relation and hence each tuple is shared between at most a pair of operators. Each such tuple is associated with a Lagrange multiplier (λ above) and hence a pair of weights (one in each operator). Each operator predicts its output relations (and all shared tuples). In FELIX, entire relations are shared (or not). Thus, the compiler generates a copy of each shared relation (cf. Section 3.2). Then, a process in FELIX, called the *Master*, compares each pair of copies of the shared output relations and computes the new Lagrange multipliers. If both copies agree on a tuple, then the Lagrange multiplier for that tuple does not change. If a tuple is present in one copy, but not the other, then the Lagrange multiplier is updated by the Master (in the example, this updates the weights in each program). When a weight changes, the amount that the weight changes is determined by a standard technique in this area (gradient descent [47, p. 174]). The process of predicting and comparing the relations is typically repeated several times.

5 Related Work

There is a trend to build rich applications that fuse the output of information extractions with increasingly sophisticated background knowledge [24, 40, 44, 45, 48]. We follow on this line of work. One distinction is that while the goal of prior work is to explore the effectiveness of different correlation structures to particular applications, our goal is to support application development by scaling up existing statistical reasoning frameworks that provide ease and flexibility of a general-purpose language combined with the high quality and efficiency of specialized algorithms.

In StatSnowball [48], Zhu et al. demonstrate high quality results of an MLN-based approach. To address the scalability issue of generic MLN inference, they make additional independence assumptions in their IE tasks. In contrast, the goal of FELIX is to automatically scale up statistical inference given the same input program. There is work on improving the performance of MLNs with alternative inference algorithms [30]. In contrast, the approach we study here moves away from the monolithic, one-algorithm inference paradigm. Theobald et al. [41] design specialized MaxSAT algorithms that efficiently solve a family of MLN programs. In contrast, we study how to scale MLN inference without limiting its expressive power.

FELIX specializes to MLNs. There are, however, other general-purpose statistical frameworks such as PRMs [14], BLOG [21], Factorie [19], and PrDB [36]. Our hope is that the techniques we develop here could apply equally well to these other general-purpose approaches. We choose MLNs because of our work on the Machine Reading project, and because they have been successfully applied to a broad range of text-based applications: natural language processing [32], ontology matching [45], and information extraction [48].

This work builds on our recent work [22] on scaling up (monolithic) MLN inference, where we built a system called TUFFY that achieves orders of magnitude scale-up and speed-up by using an RDBMS (instead
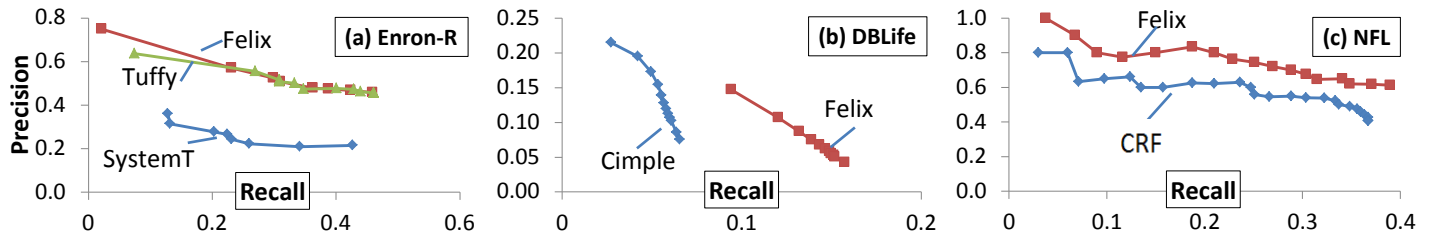
Figure 5: High-level Quality Results of Different IE Systems. Axes are zoomed into interesting regions.

of hand-coded nested loop join) for relational query processing and data partitioning. However, the scalability of TUFFY is still limited by the fact that it performs monolithic MLN inference. For example, TUFFY still has quadratic space requirement when the program has coreference subtasks. More directly, TUFFY is an operator in FELIX.

Our materialization tradeoff strategy is related to view materialization and selection [15, 38] in the context of data warehousing. However, our problem setting is different: we consider only batch processing so that we do not consider maintenance cost. The idea of lazy-eager tradeoff in view materialization or query answering has also been applied to probabilistic databases [46]. Their goal is efficiently maintaining the intermediate results, rather than choosing a materialization strategy. Similar in spirit to our approach is Sprout [23], which considers lazy-versus-eager plans for when to apply confidence computation, but does not consider an operator-based approach to inference.

Decomposition is used as the basis to tie together inference problems (say discriminative and generative models) to improve specific tasks, e.g., parsing [33]. In previous approaches one had to manually construct this decomposition. In contrast, we propose to perform the decomposition automatically (inspired by a standard relational query compiler).

The XLog framework [37] allows black boxes (e.g., Perl scripts) to be called from datalog programs (simulating table functions). Our approach differs in many key aspects: (1) FELIX uses a single language (Markov Logic) and so our operators are not black boxes rather they have a formal semantics, (2) we discover specialized routines automatically, and (3) we consider materialization trade offs.

6 Experiments

Although FELIX can be applied in more general settings than information extraction (IE), we validate our approach on such tasks. This choice allows us to compare our approach with state-of-the-art, rule-based systems for IE. These IE systems require tuning, rules, etc, and so to be as fair as possible, we run each IE system on a dataset-program pair on which they have been validated in the literature. Additionally, we compare our approach with prior statistical frameworks. We confirm previous results that such statistical approaches have high quality on small datasets (thousands of documents), but these approaches are not able to scale to large datasets (hundreds of thousands of documents). However, using the ideas in FELIX we are able to scale to these larger data sets. The rule-based approaches can scale, but cannot take advantage of sophisticated statistical reasoning and so have lower-quality extractions.

Finally, we validate our individual contributions to demonstrate that the operator-based framework and the optimization for materialization strategies are crucial to our higher scalability compared to previous approaches.[12]

Datasets and Tasks Table 4 contains statistics about the three datasets that we select: (1) **Enron**, where the task is to identify person mentions and associated phone numbers in the Enron email dataset. There are two versions of Enron: Enron-random (**Enron-R**)[13] is a subset of Enron emails that has manually annotated

---

	Enron-R	Enron	DBLife	NFL
raw data size	1.6MB	700MB	300MB	20MB
#documents	680	225K	22K	1.1K
#mentions	486	2,500K	700K	100K

Table 4: Dataset statistics

person-phone ground truth; and **Enron**[14] is the full dataset. We use Enron-R for quality assessment, and Enron for performance evaluation. (2) **DBLife**, where the task is to extract persons, organizations, and affiliation relationships between them from a collection of academic webpages. For **DBLife**, we do not have ground truth; instead, we use the ACM author profile data (`http://www.acm.org/membership/author_pages`) as ground truth. (3) **NFL**, where the task is to extract football game results from sports news articles. The NFL dataset is a testbed for an ongoing DARPA Machine Reading project, the ground truth is manually extracted.

Competitors and Methodology To make the comparison as fair as possible, we run the rule-based systems on datasets for which their creators developed programs (rather than writing our own programs in their frameworks). On DBLife, we compare against the rule-based system CIMPLE that is behind the DBLife web portal [12]. On Enron, we compare against SYSTEMT from IBM.[15] Our rules for the person-phone task are those described in two papers on SYSTEMT [18, 20]. To run the MLN-based systems, we create the corresponding MLN programs in a simple way: Each rule-based program is essentially equivalent to an MLN-based program (without weights). As these rule-based programs have ad-hoc reasoning to determine relationships and coreference between entities, we simply replace these rules by a simple statistical variant. For example, SYSTEMT says that if a person and a phone occur within a fixed window size they are associated; in the corresponding MLN we have several rules for each window-size weighted by their size. On **NFL**, we use conditional random fields (CRF), a popular statistical IE approach implemented by CRF++[16], as our baseline.

To compare with alternate implementations of MLNs, we also experiment with two state-of-the-art MLN implementations: (1) ALCHEMY, the reference implementation for MLNs, and (2) TUFFY, an RDBMS-based implementation of MLNs [22].

Experimental Setup ALCHEMY is implemented in C++. TUFFY and FELIX are both implemented in Java and use PostgreSQL 9.0.4. FELIX uses TUFFY as a library. All experiments are run on a RHEL5 workstation with two 2.67GHz Intel Xeon CPUs (24 total cores) and 24 GB of RAM.
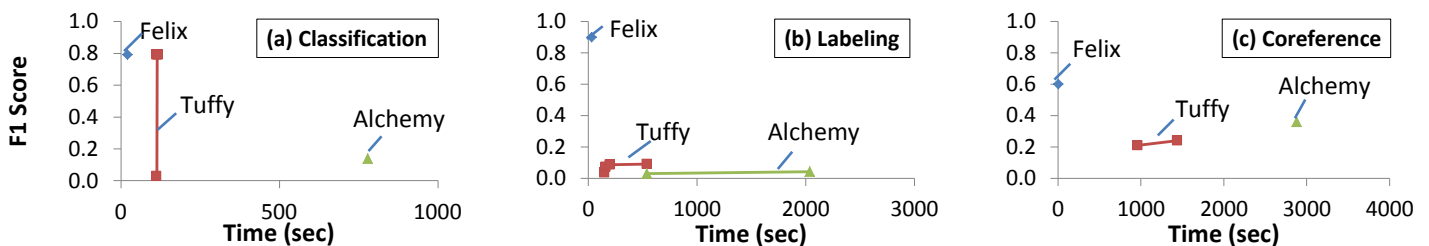
Figure 6: Comparison of FELIX, TUFFY, and ALCHEMY on specialized operators.

6.1 High-level Quality and Performance

We empirically validate that FELIX can efficiently execute complex statistical inference programs results with state-of-the-art quality. Furthermore, we show that monolithic MLN systems such as TUFFY or ALCHEMY do

---

[14]`http://bailando.sims.berkeley.edu/enron_email.html`

[15]`http://www.almaden.ibm.com/cs/projects/systemt/`

[16]`http://crfpp.sourceforge.net`

not scale when the program contains challenging subtasks such as coref. To support these claims, we compare the performance and quality of different MLN inference engines (Tuffy, Alchemy, and Felix) together with state-of-the-art approaches (both rule-based and learning-based) on the datasets listed above.

On each dataset, we run the three MLN engines, Felix, Alchemy, and Tuffy, for 3000 seconds and produce the marginal inference result.[17] We obtain the baseline by running the system corresponding to a given dataset to completion (SystemT for Enron-R, Cimple for DBLife and CRF for NFL).

To summarize the output quality of each system, we draw precision-recall curves: we take ranked lists of predictions from each system and measure precision/recall of top-k results while varying the number of answers returned. The quality of each system is shown in Figure 5.

On all three tasks, the quality of MLN-based systems is superior to the corresponding baseline approach. On Enron, Felix and Tuffy improve the precision of SystemT by 100% while retaining the same recall. On DBLife, Felix improves the recall by about 200% while maintaining similar precision. On NFL, Felix is able to boost the precision to above 80% from CRF's 60%. This supports prior work that it is possible to achieve state-of-the-art quality using an MLN-based approach [26, 27, 31].

Felix scales in cases when neither Tuffy nor Alchemy does. On all three programs, Alchemy either crashed after running out of memory or took longer than 6 hours, and so there are no curves for Alchemy in Figure 5. The reason is that Alchemy is a pure-memory implementation [22]. Tuffy (although it uses an RDBMS for memory management) crashes on both DBLife and NFL after consuming tens of gigabytes of disk space, and so there are no Tuffy curves for DBLife and NFL in Figures 5 (b) and (c). The reason Tuffy can run on Enron is that the input data is small – less than 500 mentions. This suggests that Felix's approach allows MLNs to be used on a wider array of datasets.

6.2 Operators Improve Quality & Scalability

Although Alchemy, Tuffy, and Felix all take in the same statistical language (Markov Logic), Felix outperforms Tuffy and Alchemy in both quality and scalability. The results below support our central technical claim that Felix's operator-based framework is crucial to the improved quality and scalability of Felix.

Quality We first demonstrate that Felix's specialized algorithms outperform monolithic MLN engines in both quality and performance when solving subtasks. To evaluate this claim, we run Felix, Tuffy and Alchemy on three MLN programs that each contain only one of the following subtasks: classification, labeling, and coreference. We run these three systems on a subset of the Cora dataset [18] for coref task; on a subset of the CoNLL 2000 chunking dataset [19] for labeling and classification tasks. On each of these programs, Felix's plan is a single-operator task.

We run Felix, Tuffy, Alchemy on these programs and plot the quality-time curves in Figure 6. As Figure 6 illustrates, while it always takes less than 1 minute for Felix to finish these specialized operators, Tuffy and Alchemy take longer. Moreover, the quality of Felix is higher than Tuffy and Alchemy. This is not surprising because Felix can achieve the exact optimal solution for labeling and classification operators, and a nearly optimal approximation for Coref, while Tuffy and Alchemy rely on a general-purpose SAT counting algorithm. In addition to achieving higher quality, Felix is faster than either of the other approaches.

Scalability To demonstrate that operators improve the scalability of Felix, we consider three variants of Felix where we turn off the ability of Felix to recognize an operator:

Felix\CoRef, **Felix\Label**, and **Felix\Classify**.

---

[17]After 3000 seconds, we observed that the quality of each system does not change significantly.

[18]http://alchemy.cs.washington.edu/data/cora

[19]http://www.cnts.ua.ac.be/conll2000/chunking/

Felix\CoRef means we do not detect coref operators in Felix, while we still detect Labeling and Classification operators. Felix\Label and Felix\Classify are defined similarly. We run these three variants on Enron-R, DBLife, and NFL datasets to compare their quality and performance with Felix. Table 5 shows a summary of whether these three variants scale, where "N/A" means the removed operator does not appear in the program (so the result did not change), "CRASH" means the corresponding version crashed or did not produce any results in 6 hours, "SCALES" means the corresponding version produces results. In the two cases where Felix can scale, it achieves similar quality to the operator-based version of Felix.

	Enron-R	DBLife	NFL
Felix\CoRef	SCALES	CRASH	CRASH
Felix\Label	N/A	N/A	CRASH
Felix\Classify	N/A	SCALES	N/A

Table 5: Scalability without an operator in Felix

From Table 5 we see that the three specialized operators in Felix are crucial to its scalability: without these operators, Felix cannot scale in some cases. One critical reason is that operators may reduce the size of groundings needed to perform inference significantly. For example, on the NFL dataset for one rule Felix generates only 1900 groundings; in contrast, we estimate that Felix\CoRef needs to generate more than 26 million ground facts.

6.3   Cost-based Materialization is Critical

We validate that the cost-based materialization tradeoff in Felix produces strategies that outperform both eager and lazy materialization approaches. We focus on the person coreference operator on the DBLife dataset and subsets of the Enron datasets. We compare the performance of different strategies: 1) **Eager**, where all data-movement operators are evaluated eagerly; 2) **Lazy**, where all data-movement operators are evaluated lazily; 3) **Opt**, where Felix decides the materialization strategy for each data-movement operator based on the cost model in Section 4.2.

	DBL	E-5k	E-20k	E-50k	E-100k
Eager	40 min	83 sec	15 min	134 min	641 min
Lazy	4 min	42 sec	5 min	22 min	78 min
Opt	4 min	29 sec	2 min	7 min	25 min

Table 6: Performance comparison of different materialization strategies for person coreference.

In Table 6, we show the performance of an eager approach, a lazy approach, and Felix's optimizer for materializing just the data-movement operators for coref. Here, **E-xk** for $x \in \{5, 20, 50, 100\}$ refers to a randomly selected subset of xk emails in the Enron corpus. We observe that the performance of the eager materialization strategy suffers as the dataset size increases. The lazy strategy performs much better, and the cost-based approach can further achieve 2-3X speedup. This demonstrates that our cost-based materialization tradeoff is crucial to the efficiency of Felix.

7   Conclusion and Future Work

We present our approach to inference in Markov Logic that is based on the idea that there are common subtasks in many text-processing tasks. In some tasks, these common operations are a performance or extraction-accuracy bottleneck. To ameliorate this bottleneck, we propose to use specialized algorithms for each task. To support this proposal, we develop Felix whose key technical contributions are an architecture that allows specialized algorithms to be encapsulated as operators, a compiler to find these specialized algorithms automatically, and a data-movement optimizer built into an RDBMS. Using these techniques, we demonstrate that Felix is able

to scale to complex IE programs on large datasets and generates results that have higher quality than state-of-the-art IE approaches. This result makes us optimistic that the techniques in FELIX have broader applications. One avenue of future work is to understand whether or not the operator-based approach can be applied to a wider set of applications and to other portions of the text-processing pipeline, e.g., feature extraction and weight learning, which may involve adding both new logical operators and physical implementations.

References

[1] S. Abiteboul and R. Hull. Data functions, datalog and negation. In *SIGMOD*, 1988.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison Wesley, 1995.

[3] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *JACM*, 2008.

[4] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using Dedupalog. In *ICDE 2009*.

[5] D. Bertsekas and J. Tsitsiklis. *Parallel and distributed computation: Numerical methods.* Prentice-Hall, 1989.

[6] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, 2007.

[7] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, New York, 2004.

[8] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[9] S. Chaudhuri and M. Vardi. On the complexity of equivalence between recursive and nonrecursive datalog programs. In *PODS*, 1994.

[10] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *ACL 2010*.

[11] R. Cowell, D. Philip, S. Lauritzen, and D. Spiegelhalter. *Probabilistic Networks and Expert Systems.* Springer, 2003.

[12] P. DeRose, W. Shen, , F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A community information management platform for the database research community. In *CIDR 2007*.

[13] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 1969.

[14] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *IJCAI*, 1999.

[15] H. Gupta and I. Mumick. Selection of views to materialize in a data warehouse. *TKDE*, 2005.

[16] A. Klug. On conjunctive queries containing inequalities. *J. ACM*, 1988.

[17] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[18] B. Liu, L. Chiticariu, V. Chu, H. Jagadish, and F. Reiss. Automatic rule refinement for information extraction. *VLDB*, 3(1), 2010.

[19] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS*, 2009.

[20] E. Michelakis, R. Krishnamurthy, P. Haas, and S. Vaithyanathan. Uncertainty management in rule-based information extraction systems. In *SIGMOD*, 2009.

[21] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *IJCAI*, 2005.

[22] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. In *VLDB 2011*.

[23] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, 2009.

[24] M. Paşca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Names and similarities on the web: Fact extraction in the fast lane. In *ACL*, 2006.

[25] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI 2007*.

[26] H. Poon and P. Domingos. Joint unsupervised coreference resolution with Markov Logic. In *EMNLP 2008*.

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

[28] R. Ramakrishnan and J. Ullman. A survey of deductive database systems. *J. Logic Programming*, 1995.

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

[30] S. Riedel. Cutting Plane MAP Inference for Markov Logic. In *SRL 2009*.

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

[32] S. Riedel and I. Meza-Ruiz. Collective semantic role labeling with Markov logic. In *CoNLL 2008*.

[33] A. M. Rush, D. Sontag, M. Collins, and T. Jaakkola. On dual decomposition and linear programming relaxations for natural language processing. In *EMNLP*, 2010.

[34] H. Schmid. Improvements in part-of-speech tagging with an application to German. *NLP Using Very Large Corpora*, 1999.

[35] J. Seib and G. Lausen. Parallelizing datalog programs by generalized pivoting. In *PODS*, 1991.

[36] P. Sen, A. Deshpande, and L. Getoor. PrDB: Managing and exploiting rich correlations in probabilistic databases. *J. VLDB*, 2009.

[37] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[38] A. Shukla, P. Deshpande, and J. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, 1998.

[39] S. Singh, K. Schultz, and A. McCallum. Bi-directional joint inference for entity resolution and segmentation using imperatively-defined factor graphs. 2009.

[40] F. Suchanek, M. Sozio, and G. Weikum. SOFIE: A self-organizing framework for information extraction. In *WWW*, 2009.

[41] M. Theobald, M. Sozio, F. Suchanek, and N. Nakashole. URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules. *MPI Technical Report*, 2010.

[42] J. Ullman. Implementation of logical query languages for databases. *TODS*, 1985.

[43] M. Wainwright and M. Jordan. *Graphical Models, Exponential Families, and Variational Inference*. Now Publishers Inc., Hanover, 2008.

[44] R. Wang and W. Cohen. Iterative set expansion of named entities using the web. In *ICDM*, 2008.

[45] D. Weld, R. Hoffmann, and F. Wu. Using Wikipedia to bootstrap open information extraction. *SIGMOD Record*, 2009.

[46] M. Wick, A. McCallum, and G. Miklau. Scalable Probabilistic Databases with Factor Graphs and MCMC. *VLDB*, 3(1), 2010.

[47] L. Wolsey. *Integer Programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1998.

[48] J. Zhu, Z. Nie, X. Liu, B. Zhang, and J. Wen. Statsnowball: A statistical approach to extracting entity relationships. In *WWW*, 2009.

A   Notations

Table 7 defines some common notation that is used in the following sections.

Notation	Definition
$a, b, \ldots, \alpha, \beta, \ldots$	Singular (random) variables
$\boldsymbol{a}, \boldsymbol{b}, \ldots, \boldsymbol{\alpha}, \boldsymbol{\beta}, \ldots$	Vectorial (random) variables
$\boldsymbol{\mu}' \cdot \boldsymbol{\nu}$	Dot product between vectors
$|\boldsymbol{\mu}|$	Length of a vector or size of a set
$\boldsymbol{\mu}_i$	i^{th} element of a vector
$\hat{\alpha}, \hat{\boldsymbol{\alpha}}$	A value of a variable

Table 7: Notations

B   Theoretical Background of the Operator-based Approach

In this section, we discuss the theoretical underpinning of FELIX's operator-based approach to MLN inference. Recall that FELIX first decomposes an input MLN program based on a predefined set of operators, instantiates those operators with code selection, and then executes the operators using ideas from dual decomposition. We first justify our choice of specialized subtasks (i.e., Classification, Sequential Labeling, and Coref) in terms of two compilation soundness and language expressivity properties:

1. Given an MLN program, the subprograms obtained by FELIX's compiler indeed encode specialized subtasks such as classification, sequential labeling, and coref.

2. MLN as a language is expressive enough to encode all possible models in the exponential family of each subtask type; specifically, MLN subsumes logistic regression (for classification), conditional random fields (for labeling), and correlation clustering (for coref).

We then describe how dual decomposition is used to coordinate the operators in FELIX for both MAP and marginal inference while maintaining the semantics of MLNs.

B.1 Consistent Semantics

B.1.1 MLN Program Solved as Subtasks

In this section, we show that the decomposition of an MLN program produced by FELIX's compiler indeed corresponds to the subtasks defined in Section 3.1.

Classification Suppose a classification operator for a query relation $R(k, v)$ consists of key-constraint hard rules together with rules $r_1, ..., r_t$ (with weights $w_1, ..., w_t$) [20]. As per FELIX's compilation procedure, the following holds: 1) $R(k, v)$ has a key constraint (say k is the key); and 2) none of the selected rules are recursive with respect to R.

Let k_0 be a fixed value of k. Since k is a possible-world key for $R(k, v)$, we can partition the set of all possible worlds into sets based on their v for $R(k_0, v)$ (and whether there is any value v make $R(k, v)$ true). Let $\mathcal{W}_{v_i} = \{W \mid W \models R(k_0, v_i)\}$ and $\mathcal{W}_\perp$ where $R(k_0, v)$ is false for all v. Define $Z(\mathcal{W}) = \sum_{w \in \mathcal{W}} \exp\{-cost(w)\}$. Then according to the semantics of MLN,

$$\Pr[R(k, v_0)] = \frac{Z(\mathcal{W}_{v_0})}{Z(\mathcal{W}_\perp) + \sum_{v \in \mathbb{D}} Z(\mathcal{W}_v)}$$

It is immediate from this that each class is disjoint. It is also clear that, conditioned on the values of the rule bodies, each of the R are independent.

Sequence Labeling Suppose a labeling operator outputs a relation $R(k, v)$ and consists of hard-constraint rules together with ground rules $r_1, ... r_t$ (with weights $w_1, ..., w_t$). As per FELIX's compilation procedure, the following holds:

- $R(k, v)$ has a key constraint (say k is the key);

- The rules r_i satisfy the TrREC property: For rules like Eqn. 2, T's transitive closure is a strict partial order and T satisfies a key constraint on k.

Consider the following graph: the nodes are all possible values for the key k and there is an edge (k, k') if k appears in the body of k'. Every node in this graph has outdegree at most 1. Now suppose there is a cycle: But this contradicts the definition of a strict partial order. In turn, this means that this graph is a forest. Then, we identify this graph with a graphical model structure where each node is a random variable with domain $\mathbb{D}$. This is a tree-structured Markov random field. This justifies the rules used by FELIX's compiler for identifying labeling operators. Again, conditioned on the rule bodies any grounding is a tree-shaped graphical model.

Coreference Resolution A coreference resolution subtask involving variables $y_1, ... y_n$ infers about an equivalent relation $R(y_i, y_j)$. The only requirement of this subtask is that the result relation $R(., .)$ be reflexive, symmetric and transitive. FELIX ensures these properties by detecting corresponding hard rules directly.

B.1.2 Subtasks Represented as MLN programs

We start by showing that all probabilistic distributions in the discrete exponential family can be represented by an equivalent MLN program. Therefore, if we model the three subtasks using models in the exponential family, we can express them as an MLN program. Fortunately, for each of these subtasks, there are popular exponential family models: 1) Logistic Regression (LR) for Classification, 2) Conditional Random Filed (CRF) for Labeling and 3) Correlation Clustering for Coref. [21]

---

[20] For simplicity, we assume that these t rules are ground formulas. It is easy to show that grounding does not change the property of rules.

[21] We leave the discussion of models that are not explicitly in exponential family to future work.

Definition B.1 (Exponential Family). *We follow the definition in [43]. Given a vector of binary random variables $\boldsymbol{x} \in \mathcal{X}$, let $\boldsymbol{\phi} : \mathcal{X} \to \{0,1\}^d$ be a binary vector-valued function. For a given $\boldsymbol{\phi}$, let $\boldsymbol{\theta} \in \mathbb{R}^d$ be a vector of real number parameters. The exponential family distribution over $\boldsymbol{x}$ associated with $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ is of the form:*

$$\Pr_{\boldsymbol{\theta}}[\boldsymbol{x}] = \exp\{-\boldsymbol{\theta} \cdot \boldsymbol{\phi}(\boldsymbol{x}) - A(\boldsymbol{\theta})\},$$

where $A(\boldsymbol{\theta})$ is known as log partition function: $A(\boldsymbol{\theta}) = \log \sum_{\boldsymbol{x} \in \mathcal{X}} \exp\{-\boldsymbol{\theta} \cdot \boldsymbol{\phi}(\boldsymbol{x})\}$.

This definition extends to multinomial random variables in a straightforward manner. For simplicity, we only consider binary random variables in this section.

Example 1 Consider a textbook logistic regressor over a random variable $x \in \{0,1\}$:

$$\Pr[x = 1] = \frac{1}{1 + \exp\{\sum_i -\beta_i f_i\}},$$

where $f_i \in \{0,1\}$'s are known as features of x and β_i's are *regression coefficients* of f_i's. This distribution is actually in the exponential family: Let $\boldsymbol{\phi}$ be a binary vector-valued function whose i^{th} entry equals to $\phi_i(x) = (1 - x)f_i$. Let $\boldsymbol{\theta}$ be a vector of real numbers whose i^{th} entry $\boldsymbol{\theta}_i = \beta_i$. One can check that

$$\Pr[x = 1] = \frac{\exp\{-\boldsymbol{\theta} \cdot \boldsymbol{\phi}(1)\}}{\exp\{-\boldsymbol{\theta} \cdot \boldsymbol{\phi}(1)\} + \exp\{-\boldsymbol{\theta} \cdot \boldsymbol{\phi}(0)\}}$$

$$= \frac{1}{1 + \exp\{\sum_i -\beta_i f_i\}}$$

The exponential family has a strong connection with the maximum entropy principle and graphic models. For all the three tasks we are considering, i.e., classification, labeling and coreference, there are popular exponential family models for each of them.

Proposition B.1. *Given an exponential family distribution over $\boldsymbol{x} \in \mathcal{X}$ associated with $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, there exists an MLN program Γ that defines the same probability distribution as $\Pr_{\boldsymbol{\theta}}[\boldsymbol{x}]$. The length of the formula in Γ is at most linear in $|\boldsymbol{x}|$, and the number of formulas in Γ is at most exponential in $|\boldsymbol{x}|$.*

Proof. Our proof is by construction. Each entry of $\boldsymbol{\phi}$ is a binary function $\phi_i(\boldsymbol{x})$, which partitions $\mathcal{X}$ into two subsets: $\mathcal{X}_i^+ = \{\boldsymbol{x} | \phi_i(\boldsymbol{x}) = 1\}$ and $\mathcal{X}_i^- = \{\boldsymbol{x} | \phi_i(\boldsymbol{x}) = 0\}$. If $\theta_i \geq 0$, for each $\hat{\boldsymbol{x}} \in \mathcal{X}_i^+$, introduce a rule:

$$\theta_i \quad \bigvee_{1 \leq j \leq |\boldsymbol{x}|} R(x_j, 1 - \hat{x}_j).$$

If $\theta_i < 0$, for each $\hat{\boldsymbol{x}} \in \mathcal{X}_i^+$, insert a rule:

$$-\theta_i \quad \bigwedge_{1 \leq j \leq |\boldsymbol{x}|} R(x_j, \hat{x}_j).$$

We add these rules for each $\phi_i(.)$, and also add the following hard rule for each variable x_i:

$$\infty \quad R(x_i, 0) \quad <=> \quad \neg R(x_i, 1).$$

It is not difficult to see $\Pr[\forall x_i, R(x_i, \hat{x}_i) = 1] = \Pr_{\boldsymbol{\theta}}[\hat{\boldsymbol{x}}]$. In this construction, each formula has length $|\boldsymbol{x}|$ and there are $\sum_i(|\mathcal{X}_i| + 1)$ formulas in total, which is exponential in $|\boldsymbol{x}|$ in the worst case. $\qquad\square$

Similar constructions apply to the case where $\boldsymbol{x}$ is a vector of multinomial random variables.

We then show that Logistic Regression, Conditional Random Field and Correlation Clustering all define probability distributions in the discrete exponential family, and the number of formulas in their equivalent MLN program Γ is polynomial in the number of random variables.

Logistic Regression In Logistic Regression, we model the probability distribution of Bernoulli variable y conditioned on $x_1, ..., x_k \in \{0, 1\}$ by

$$\Pr[y = 1] = \frac{1}{1 + \exp\{-(\beta_0 + \sum_i \beta_i x_i)\}}$$

Define $\phi_i(y) = (1 - y)x_i$ ($\phi_0(y) = 1 - y$) and $\theta_i = \beta_i$, we can see $\Pr[y = 1]$ is in the exponential family defined as in Definition B.1. For each $\phi_i(y)$, there is only one y that can get positive value from ϕ_i, so there are at most $k + 1$ formulas in the equivalent MLN program.

Conditional Random Field In Conditional Random Field, we model the probability distribution using a graph $G = (V, E)$ where V represents the set of random variables $\boldsymbol{y} = \{y_v : v \in V\}$. Conditioned on a set of random variables $\boldsymbol{x}$, CRF defines the distribution:

$$\Pr[\boldsymbol{y}|\boldsymbol{x}] \propto \exp\{ \sum_{v \in V, k} \lambda_k f_k(v, y_v, \boldsymbol{x})$$
$$+ \sum_{(v_1, v_2) \in E, l} \mu_l g_l((v_1, v_2), y_{v_1}, y_{v_2}, \boldsymbol{x})\}$$

This is already in the form of exponential family. Because each function $f_k(v, -, \boldsymbol{x})$ or $g_l((v_1, v_2), -, -, \boldsymbol{x})$ only relies on 1 or 2 random variables, the resulting MLN program has at most $O(|E| + |V|)$ formulas. In the current prototype of FELIX, we only consider linear chain CRFs, where $|E| = O(|V|)$.

Correlation Clustering Correlation clustering is a form of clustering for which there are efficient algorithms that have been shown to scale to instances of the coref problem with millions of mentions. Formally, correlation clustering treats the coref problem as a graph partitioning problem. The input is a weighted undirected graph $G = (V, f)$ where V is the set of mentions with weight function $f : V^2 \to \mathbb{R}$. The goal is to find a partition $\mathcal{C} = \{C_i\}$ of V that minimizes the *disagreement cost*:

$$cost_{cc}(\mathcal{C}) = \sum_{\substack{(v1, v2) \in V^2 \\ v_1 \neq v_2 \\ \exists C_i, v_1 \in C_i \land v_2 \in C_i \\ f(u,v) < 0}} |f(v_1, v_2)| + \sum_{\substack{(v1, v2) \in V^2 \\ v_1 \neq v_2 \\ \exists C_i, v_1 \in C_i \land v_2 \notin C_i \\ f(u,v) > 0}} |f(v_1, v_2)|$$

We can define the probability distribution over $\mathcal{C}$ similarly as MLN:

$$\Pr[\mathcal{C}] \propto \exp\{-cost_{cc}(\mathcal{C})\}$$

Specifically, let the binary predicate $coref(v_1, v_2)$ indicate whether $v_1 \neq v_2 \in V$ belong to the same cluster. First introduce three hard rules enforcing the reflexivity, symmetry, and transitivity properties of $coref$. Next, for each $v_1 \neq v_2 \in V$, introduce a singleton rule $coref(v_1, v_2)$ with weight $f(v_1, v_2)$. It's not hard to show that the above distribution holds for this MLN program.

B.2 Dual Decomposition for MAP and Marginal Inference

In this section, we formally describe the dual decomposition framework used in FELIX to coordinate the operators. We start by formalizing MLN inference as an optimization problem. Then we show how to apply dual decomposition on these optimization problems.

B.2.1 Problem Formulation

Suppose an MLN program Γ consists of a set of ground MLN rules $\mathcal{R} = \{r_1, ..., r_m\}$ with weights $(w_1, ..., w_m)$. Let $X = \{x_1, ..., x_n\}$ be the set of boolean random variables corresponding to the ground atoms occurring in Γ. Each MLN rule r_i introduces a function ϕ_i over the set of random variables $\pi_i \subseteq X$ mentioned in r_i: $\phi_i(\pi_i) = 1$ if r_i is violated and 0 otherwise. Let $\boldsymbol{w}$ be a vector of weights. Define vector $\boldsymbol{\phi}(X) = (\phi_1(\pi_1), ..., \phi_m(\pi_m))$. Given a possible world $\boldsymbol{x} \in 2^X$, the cost can be represented:

$$cost(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x})$$

Suppose FELIX decides to solve Γ with t operators $O_1, ..., O_t$. Each operator O_i contains a set of rules $\mathcal{R}_i \subseteq \mathcal{R}$. The set $\{\mathcal{R}_i\}$ forms a partition of $\mathcal{R}$. Let the set of random variables for each operator be $X_i = \cup_{r_j \in \mathcal{R}_i} \pi_j$. Let $n_i = |X_i|$. Thus, each operator O_i essentially solves the MLN program defined by random variables X_i and rules $\mathcal{R}_i$. Given $\boldsymbol{w}$, define $\boldsymbol{w}^i$ to be the weight vector whose entries equal $\boldsymbol{w}$ if the corresponding rule appears in $\mathcal{R}_i$ and 0 otherwise. Because $\mathcal{R}_i$ forms a partition of $\mathcal{R}$, we know $\sum_i \boldsymbol{w}^i = \boldsymbol{w}$. For each operator O_i, define an n-dim vector $\boldsymbol{\mu}_i(X)$, whose j^{th} entry equals x_j if $x_j \in X_i$ and 0 otherwise. Define n-dim vector $\boldsymbol{\mu}(X)$ whose j^{th} entry equals x_j. Similarly, let $\boldsymbol{\phi}(X_i)$ be the projection of $\boldsymbol{\phi}(X)$ onto the rules in operator O_i.

Example 2 We use the two sets of rules for classification and labeling in Section 2.2 as a running example. For a simple sentence *Packers win.* in a fixed document D which contains two phrases $P_1 = $ "Packers" and $P_2 = $ "win", we will get the following set of ground formulae [22]:

∞	$\texttt{label}(p, l1), \texttt{label}(p, l2) => l1 = l2$	(r_{l1})
10	$\texttt{next}(P_1, P_2), \texttt{token}(P_2, \text{`wins'}) => \texttt{label}(P_1, W)$	(r_{l2})
1	$\texttt{label}(P_1, W), \texttt{next}(P_1, P_2) =>!\texttt{label}(P_2, W)$	(r_{l3})
10	$\texttt{label}(P_1, W), \texttt{referTo}(P_1, GreenBay) => \texttt{winner}(GreenBay)$	(r_{c1})
10	$\texttt{label}(P_1, L), \texttt{referTo}(P_1, GreenBay) => !\texttt{winner}(GreenBay)$	(r_{c2})

After compilation, FELIX would assign r_{l1}, r_{l2} and r_{l3} to a labeling operator O_L, and r_{c1} and r_{c2} to a classification operator O_C. For each of $\{\texttt{winner}(GreenBay), \texttt{label}(P_1, W), \texttt{label}(P_1, L), \texttt{label}(P_2, W), \texttt{label}(P_2, L)\}$ we have a binary random variable associated with it. Each rule introduces a function ϕ, for example, the function ϕ_{l2} introduced by r_{l2} is:

$$\phi_{l2}(\texttt{label}(P_1, W)) = \begin{cases} 1 & \text{if } \texttt{label}(P_1, W) = \text{False} \\ 0 & \text{if } \texttt{label}(P_1, W) = \text{True} \end{cases}$$

The labeling operator O_L essentially solves the MLN program with variables $X_L = \{\texttt{label}(P_1, W), \texttt{label}(P_1, L), \texttt{label}(P_2, W), \texttt{label}(P_2, L)\}$ and rules $\mathcal{R}_L = \{r_{l1}, r_{l2}, r_{l3}\}$. Similarly O_C solves the MLN program with variables $X_C = \{\texttt{winner}(GreenBay), \texttt{label}(P_1, W) \texttt{label}(P_1, L)\}$ and rules $\mathcal{R}_C = \{r_{c1}, r_{c2}\}$. Note that these two operators share the variables $\texttt{label}(P_1, W)$ and $\texttt{label}(P_1, L)$.

B.2.2 MAP Inference

MAP inference in MLNs is to find an assignment $\boldsymbol{x}$ to X that minimizes the cost:

$$\min_{\boldsymbol{x} \in \{0,1\}^n} \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}). \tag{3}$$

Each operator O_i performs MAP inference on X_i:

$$\min_{\boldsymbol{x}_i \in \{0,1\}^{n_i}} \boldsymbol{w}^i \cdot \boldsymbol{\phi}(\boldsymbol{x}_i). \tag{4}$$

---

[22] For r_{l1}, $p \in \{P_1, P_2\}, l_i \in \{W, L\}$.

Our goal is to reduce the problem represented by Eqn. 3 into subproblems represented by Eqn. 4. Eqn. 3 can be rewritten as

$$\min_{\boldsymbol{x} \in \{0,1\}^n} \sum_{1 \leq i \leq t} \boldsymbol{w}^i \cdot \boldsymbol{\phi}(\boldsymbol{x}_i).$$

Clearly, the difficulty lies in that, for $i \neq j$, X_i and X_j may overlap. Therefore, we introduce a copy of variables for each O_i: X_i^C. Eqn. 3 now becomes:

$$\min_{\boldsymbol{x}_i^C \in \{0,1\}^{n_i}, \boldsymbol{x}} \quad \sum_i \boldsymbol{w}^i \cdot \boldsymbol{\phi}(\boldsymbol{x}_i^C)$$

$$s.t. \quad \forall i \quad \boldsymbol{x}_i^C = \boldsymbol{x}. \tag{5}$$

The Lagrangian of this problem is:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{x}_1^C, ..., \boldsymbol{x}_t^C, \boldsymbol{\nu}_1, ..., .\boldsymbol{\nu}_t)$$
$$= \sum_i \boldsymbol{w}^i \cdot \boldsymbol{\phi}(\boldsymbol{x}_i^C) + \boldsymbol{\nu}_i \cdot (\boldsymbol{\mu}_i(\boldsymbol{x}_i^C) - \boldsymbol{\mu}_i(\boldsymbol{x})) \tag{6}$$

Thus, we can relax Eqn. 3 into

$$\max_{\boldsymbol{\nu}} \left\{ \sum_i \left[\min_{\boldsymbol{x}_i \in \{0,1\}^{n_i}} \boldsymbol{w}^i \cdot \boldsymbol{\phi}(\boldsymbol{x}_i^C) + \boldsymbol{\nu}_i \cdot \boldsymbol{\mu}_i(\boldsymbol{x}_i^C) \right] - \max_{\boldsymbol{x}} \sum_i \boldsymbol{\nu}_i \cdot \boldsymbol{\mu}_i(\boldsymbol{x}) \right\}$$

The term $\max_{\boldsymbol{x}} \sum_i \boldsymbol{\nu}_i \cdot \boldsymbol{\mu}_i(\boldsymbol{x}) = \infty$ unless for each variable x_j,

$$\sum_{O_i : x_j \in X_i} \boldsymbol{\nu}_{i,j} = 0.$$

Converting this into constraints, we get

$$\max_{\boldsymbol{\nu}} \quad \left\{ \sum_i \min_{\boldsymbol{x}_i \in \{0,1\}^{n_i}} \boldsymbol{w}^i \cdot \boldsymbol{\phi}(\boldsymbol{x}_i^C) + \boldsymbol{\nu}_i \cdot \boldsymbol{\mu}_i(\boldsymbol{x}_i^C) \right\}$$

$$s.t. \quad \forall x_j \quad \sum_{O_i : x_j \in X_i} \boldsymbol{\nu}_{i,j} = 0$$

We can apply sub-gradient methods on $\boldsymbol{\nu}$. The dual decomposition procedure in FELIX works as follows:

1. Initialize $\boldsymbol{\nu}_1^{(0)}, ..., \boldsymbol{\nu}_t^{(0)}$.

2. At step k (starting from 0):

    (a) For each operator O_i, solve the MLN program consisting of: 1) original rules in this operator, which are characterized by $\boldsymbol{w}^i$; 2) additional priors on each variables in X_i, which are characterized by $\boldsymbol{\nu}_i^{(k)}$.

    (b) Get the MAP inference results $\hat{\boldsymbol{x}}_i^C$.

3. Update $\boldsymbol{\nu}_i$:
$$\boldsymbol{\nu}_{i,j}^{(k+1)} = \boldsymbol{\nu}_{i,j}^{(k)} - \lambda \left(\hat{\boldsymbol{x}}_{i,j}^C - \frac{\sum_{l : x_j \in X_l} \hat{\boldsymbol{x}}_{l,j}^C}{|\{l : x_j \in X_l\}|} \right)$$

Example 3 Consider the MAP inference on program in Example 2. As O_L and O_C share two random variables: $x_w = \texttt{label}(P_1, W)$ and $x_l = \texttt{label}(P_1, L)$, we have a copy of them for each operator: x_{w,O_L}^C, x_{l,O_L}^C for O_L; and x_{w,O_C}^C, x_{l,O_C}^C for O_C. Therefore, we have four ν: ν_{w,O_L}, ν_{l,O_L} for O_L; and ν_{w,O_C}, ν_{l,O_C} for O_C. Assume we initialize each $\nu_\_^{(0)}$ to 0 at the first step.

We start by performing MAP inference on O_L and O_C respectively. In this case, O_L will get the result:

$$x_{w,O_L}^C = 1$$
$$x_{l,O_L}^C = 0$$

O_C admits multiple possible worlds minimizing the cost; for example, it may outputs

$$x_{w,O_C}^C = 0$$
$$x_{l,O_C}^C = 0$$

which has cost 0. Assume the step size $\lambda = 0.5$. We can update ν to:

$$\nu_{w,O_L}^{(1)} = -0.25$$
$$\nu_{w,O_C}^{(1)} = 0.25$$

$$\nu_{l,O_L}^{(1)} = 0$$
$$\nu_{l,O_C}^{(1)} = 0$$

Therefore, when we use these $\nu_\_^{(1)}$ to conduct MAP inference on O_L and O_C, we are equivalently adding

$$-0.25 \ \texttt{label}(P_1, W) \quad (r_l')$$

into O_L and

$$0.25 \ \texttt{label}(P_1, W) \quad (r_c')$$

into O_C. Intuitively, one may interpret this procedure as the information that "O_L prefers $\texttt{label}(P_1, W)$ to be true" being passed to O_C via r_c'.

B.2.3 Marginal Inference

The marginal inference of MLNs aims at computing the marginal distribution (i.e., the expectation since we are dealing with boolean random variables):

$$\hat{\boldsymbol{\mu}} = \mathbb{E}_{\boldsymbol{w}}[\boldsymbol{\mu}(X)]. \tag{7}$$

The sub-problem of each operator is of the form:

$$\hat{\boldsymbol{\mu}}_O = \mathbb{E}_{\boldsymbol{w}_O}[\boldsymbol{\mu}_O(X_O)]. \tag{8}$$

Again, the goal is to use solutions for Eqn. 8 to solve Eqn. 7.

We first introduce some auxiliary variables. Recall that $\boldsymbol{\mu}(X)$ corresponds to the set of random variables, and $\boldsymbol{\phi}(X)$ corresponds to all functions represented by the rules. We create a new vector $\boldsymbol{\xi}$ by concatenating $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$: $\boldsymbol{\xi}(X) = (\boldsymbol{\mu}^T(X), \boldsymbol{\phi}^T(X))$. We create a new weight vector $\boldsymbol{\theta} = (0, ..., 0, \boldsymbol{w}^T)$ which is of the same length as $\boldsymbol{\xi}$. It is not difficult to see that the marginal inference problem equivalently becomes:

$$\hat{\boldsymbol{\xi}} = \mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{\xi}(X)]. \tag{9}$$

Similarly, we define $\boldsymbol{\theta}_O$ for operator O as $\boldsymbol{\theta}_O = (0, ..., 0, \boldsymbol{w}_O^T)$. We also define a set of $\boldsymbol{\theta}$: Θ_O, which contains all vectors with entries corresponding to random variables or cliques not appear in operator O as zero. The partition function $A(\boldsymbol{\theta})$ is:

$$A(\boldsymbol{\theta}) = \sum_{\mathcal{X}} \exp\{-\boldsymbol{\theta} \cdot \boldsymbol{\xi}(\mathcal{X})\}$$

The conjugate dual to A is:

$$A^*(\boldsymbol{\xi}) = \sup_{\boldsymbol{\theta}}\{\boldsymbol{\theta} \cdot \boldsymbol{\xi} - A(\boldsymbol{\theta})\}$$

A classic result of variational inference [43] shows that

$$\hat{\boldsymbol{\xi}} = \arg\sup_{\boldsymbol{\xi} \in \mathcal{M}}\{\boldsymbol{\theta} \cdot \boldsymbol{\xi} - A^*(\boldsymbol{\xi})\}, \tag{10}$$

where $\mathcal{M}$ is the marginal polytope. Recall that $\hat{\boldsymbol{\xi}}$ is our goal (see Eqn. 9). Similar to MAP inference, we want to decompose Eqn. 10 into different operators by introducing copies of shared variables. We first try to decompose $A^*(\boldsymbol{\xi})$. In $A^*(\boldsymbol{\xi})$, we search $\boldsymbol{\theta}$ on all possible values for $\boldsymbol{\theta}$. If we only search on a subset of $\boldsymbol{\theta}$, we can get a lower bound:

$$A^{*O}(\boldsymbol{\xi}) = \sup_{\boldsymbol{\theta} \in \Theta_O}\{\boldsymbol{\theta} \cdot \boldsymbol{\xi} - A^*(\boldsymbol{\xi})\} \leq A^*(\boldsymbol{\xi}).$$

Therefore,

$$-A^*(\boldsymbol{\xi}) \leq \sum_O -A^{*O}(\boldsymbol{\xi}).$$

We will instead finding an approximated $\hat{\boldsymbol{\xi}}$ by maximizing the following object functions:

$$\hat{\boldsymbol{\xi}} = \arg\sup_{\boldsymbol{\xi} \in \mathcal{M}}\{\boldsymbol{\theta} \cdot \boldsymbol{\xi} - \sum_O A^{*O}(\boldsymbol{\xi})\},$$

which is an upper bound of the original goal. We introduce copies of $\boldsymbol{\xi}$:

$$\hat{\boldsymbol{\xi}} = \arg\sup_{\boldsymbol{\xi}^{O_i} \in \mathcal{M}, \boldsymbol{\xi}}\{\sum_O \boldsymbol{\theta}_O \cdot \boldsymbol{\xi}^O - \sum_O A^{*O}(\boldsymbol{\xi}^O)\}$$
$$s.t. \ \boldsymbol{\xi}_e^O = \boldsymbol{\xi}_e, \forall e \in \mathcal{X}_O \cup \mathcal{R}_O, \forall O$$

The Lagrangian of this problem is:

$$\mathcal{L}(\boldsymbol{\xi}, \boldsymbol{\xi}^{O_1}, ..., \boldsymbol{\xi}^{O_t}, \boldsymbol{\nu}_1, ..., \boldsymbol{\nu}_t) = \sum_O \{\boldsymbol{\theta}_O \cdot \boldsymbol{\xi}^O - A^{*O}(\boldsymbol{\xi}^O)\}$$
$$+ \sum_i \boldsymbol{\nu}_i \cdot (\boldsymbol{\xi}^{O_i} - \boldsymbol{\xi}),$$

where $\boldsymbol{\nu}_i \in \Theta_i$, which means only the entries corresponding to random variables or cliques that appear in operator O_i are allowed to have non-zero values. We get the relaxation:

$$\min_{\boldsymbol{\nu}_i \in \Theta_i} \sum_i \sup_{\boldsymbol{\xi}^{O_i} \in \mathcal{M}}\{\boldsymbol{\theta}_i \cdot \boldsymbol{\xi}^{O_i} - A^{*O_i}(\boldsymbol{\xi}^{O_i}) + \boldsymbol{\nu}_i \cdot \boldsymbol{\xi}^{O_i}\}$$
$$- \min_{\boldsymbol{\xi}} \sum_i \boldsymbol{\nu}_i \cdot \boldsymbol{\xi}$$

Considering the $\min_{\boldsymbol{\xi}} \sum_i \boldsymbol{\nu}_i \cdot \boldsymbol{\xi}$ part. This part is equivalent to a set of constraints:

$$\sum_{O_i:x\in X_i} \boldsymbol{\nu}_{i,x} = 0, \forall x \in X$$

$$\boldsymbol{\nu}_{i,x} = 0, \forall x \notin X$$

Therefore, we are solving:

$$\min_{\boldsymbol{\nu}_i \in \Theta_i} \sum_i \sup_{\boldsymbol{\xi}^{O_i} \in \mathcal{M}} \left\{ \boldsymbol{\theta}_i \cdot \boldsymbol{\xi}^{O_i} - A^{*O_i}(\boldsymbol{\xi}^{O_i}) + \boldsymbol{\nu}_i \cdot \boldsymbol{\xi}^{O_i} \right\}$$

$$s.t., \quad \sum_{O_i:x\in X_i} \boldsymbol{\nu}_{i,x} = 0, \forall x \in X$$

$$\boldsymbol{\nu}_{i,x} = 0, \forall x \notin X$$

We can apply sub-gradient method on $\boldsymbol{\nu}_i$. The whole process used in FELIX looks like the following:

1. Initialize $\boldsymbol{\nu}_1^{(0)}, ..., \boldsymbol{\nu}_t^{(0)}$.

2. At step k (start from 0):

   (a) For each operator O_i, solve the MLN program consists of: 1) original rules in this operator, which is characterized by $\boldsymbol{\theta}_i$; 2) additional priors on each variables in $\mathcal{X}_i$, which is characterized by $\boldsymbol{\nu}_i^{(k)}$.

   (b) Get the marginal inference results $\hat{\boldsymbol{\xi}}_i^C$. (We do not need to get the marginal result on cliques because their corresponding values in $\boldsymbol{\nu}$ are always 0).

3. Update $\boldsymbol{\nu}_i^{(k+1)}$:

$$\boldsymbol{\nu}_{i,j}^{(k+1)} = \boldsymbol{\nu}_{i,j}^{(k)} - \lambda \left(\hat{\boldsymbol{\xi}}_{i,j}^C - \frac{\sum_{l:x_j \in X_l} \hat{\boldsymbol{\xi}}_{l,j}^C}{|\{l:x_j \in X_l\}|} \right)$$

Example 4 Consider the marginal inference on the case in Example 2. Similar to the example for MAP inference, we have copies of random variables: ξ_{w,O_L}^C, ξ_{l,O_L}^C for O_L; and ξ_{w,O_C}^C, ξ_{l,O_C}^C for O_C. We also have four ν: ν_{w,O_L}, ν_{l,O_L} for O_L; and ν_{w,O_C}, ν_{l,O_C} for O_C. Assume we initialize each $\nu_-^{(0)}$ to 0 at the first step.

We start by conducting marginal inference on O_L and O_C respectively. In this case, O_L will get the result:

$$\xi_{w,O_L}^C = 0.99$$

$$\xi_{l,O_L}^C = 0.01$$

while O_C will get:

$$\xi_{w,O_C}^C = 0.5$$

$$\xi_{l,O_C}^C = 0.5$$

Assume the step size $\lambda = 0.5$. We can update ν as:

$$\nu_{w,O_L}^{(1)} = -0.12$$

$$\nu_{w,O_C}^{(1)} = 0.12$$

$$\nu_{l,O_L}^{(1)} = 0.12$$

$$\nu_{l,O_C}^{(1)} = -0.12$$

Therefore, when we use these $\nu_-^{(1)}$ to conduct marginal inference on O_L and O_C, we are equivalantly adding

$$-0.12 \quad \texttt{label}(P_1, W)\,(r'_{l1})$$
$$0.12 \quad \texttt{label}(P_1, L)\ (r'_{l2})$$

into O_L and

$$0.12 \quad \texttt{label}(P_1, W)\,(r'_{c1})$$
$$-0.12 \quad \texttt{label}(P_1, L)\ (r'_{c2})$$

into O_C. Intuitively, one may interpret this procedure as the information that "O_L prefers $\texttt{label}(P_1, W)$ to be true" being passed to O_C via r'_c.

C   Additional Details of System Implementation

In this section, we provide additional details of the FELIX system. The first part of this section focuses on the compiler. We prove some complexity results of property-annotation used in the compiler and describe how to apply static analysis techniques originally used in the Datalog literature for data partitioning. Then we describe the physical implementation for each logical operator in the current prototype of FELIX. We also describe the cost model used for the materialization trade-off.

C.1   Compiler

C.1.1   Complexity Results

In this section, we first prove the decidability of the problem of annotating properties for arbitrary Datalog programs. Then we prove the $\Pi_2\mathsf{P}$-completeness of the problem of annotating $\{REF, SYM\}$ given a Datalog program without recursion.

Recursive Programs   If there is a single rule with query relation Q of the form $Q(x, y) <= Q1(x), Q2(y)$, then that $\{\text{REF}, \text{SYM}\}$ of Q is decidable if and only if $Q1$ or $Q2$ is empty or $Q1 \equiv Q2$. We assume that $Q1$ and $Q2$ are satisfiable. If there is an instance where $Q1(a)$ is true and $Q2$ is false for all values. Then there is another world (with all fresh constants) where $Q2$ is true (and does not return a). Thus, to check REF and SYM for Q, we need to decide equivalence of datalog queries. Equivalence of datalog queries is undecidable [2, ch. 12]. Since containment and boundedness for monadic datalog queries is decidable, a small technical wrinkle is that while $Q1$ and $Q2$ are of arity one (monadic) their bodies may contain other recursive (higher arity) predicates.

Complexity for Nonrecursive Program   The above section assumes that we are given an arbitrary Datalog program Γ. In this section, we show that the problem of annotating REF and SYM given a nonrecursive Datalog program is $\Pi_2\mathsf{P}$-complete. We allow inequalities in the program.

We first prove the hardness. Similar to the above section, we need to decide $Q1 \equiv Q2$. The difference is that $Q1$ and $Q2$ do not have recursions. Since our language allows us to express conjunctive queries with inequality constraints, this established $\Pi_2\mathsf{P}$ hardness [16].

We now prove the membership in $\Pi_2\mathsf{P}$. We first translate the problem of property-annotation to the containment problem of Datalog programs, which has been studied for decades [8, 16] and the complexity is in $\Pi_2\mathsf{P}$ for Datalog programs without recursions but with inequalities. We will show that, even though the rules for checking symmetric property is recursive, it can be represented by a set of non-recursive rules, therefore the classic results still hold.

We thus limit ourselves to non-recursive MLN programs. Given an MLN program Γ which is the union of conjunctive queries and a relation Q to which we will annotate properties, all hard rules related to Q can be represented as:

$$Q() : -G_1()$$
$$Q() : -G_2()$$
$$...$$
$$Q() : -G_n() \tag{P_1}$$

where each $G_i()$ contains a set of subgoals. To annotate whether a property holds for the relation $Q()$, we test whether some rules hold for all database instances I generated by the above program P_1. For example, for the symmetric property, we label $Q()$ as symmetric if and only if $Q(x,y) => Q(y,x)$ holds. We call this rule the *testing rule*. Suppose the testing rule is $Q() : -T()$, we create a new program:

$$Q() : -G_1()$$
$$Q() : -G_2()$$
$$...$$
$$Q() : -G_n() \tag{P_2}$$
$$Q() : -T()$$

Given a database D, let $P_1(D)$ be the result of applying program P_1 to D (using Datalog semantics). The testing rule holds for all $P_1(D)$ if and only if $\forall D,\ P_2(D) \subseteq P_1(D)$. In other words, P_2 is contained by P_1 ($P_2 \subseteq P_1$). For reflexive property, whose testing rule is $Q(x,x) : -\mathcal{D}(x)$ (where $\mathcal{D}()$ is the domain of x), both P_1 and P_2 are non-recursive and the checking of containment is in $\Pi_2 P$ [16].

We then consider the symmetric property, whose testing rule is recursive. This is difficult at first glance because the containment of recursive Datalog program is undecidable. However, for this special case, we can show it is much easier. For the sake of simplicity, we consider a simplified version of P_1 and P_2:

$$Q(x,y) : -G(x,y,z) \tag{P_1'}$$

$$Q(x,y) : -G(x,y,z)$$
$$Q(x,y) : -Q(y,x) \tag{P_2'}$$

We construct the following program:

$$Q(x,y) : -G(x,y,z)$$
$$Q(x,y) : -G(y,x,z) \tag{P_3}$$

It is easy to show $P_2' = P_3$, therefore, we can equivalently check whether $P_3 \subseteq P_1'$, which is in $\Pi_2 P$ since neither of the programs is recursive.

C.1.2 Patterns Used by the Compiler

FELIX exploits a set of regular expressions for property annotation. This set of regular expressions forms a best-effort compiler, which is sound but not complete. Table 8 shows these patterns. In FELIX, a pattern consists of two components – a template and a boolean expression. A template is a constraint on the "shape" of the formula. For example, one template for SYM looks like $P_1(a,b) \lor !P_2(c,d)$, which means we only consider rules whose disjunction form contains exactly two binary predicates with opposite senses. Rules that pass the template-matching are considered further using the boolean expression. If one rule passes the template-matching step, we can have a set of assignments for each predicate P and variable $a, b, ...$. The boolean expression is a first order logic formula on the assignment. For example, the boolean expression for the above template is $(a = d) \land (b = c) \land (P_1 = P_2)$, which means the assignment of P_1 and P_2 must be the same, and the assignment of variables a, b, c, d must satisfy $(a = d) \land (b = c)$. If there is an assignment that satisfies the boolean expression, we say this Datalog rule *matches* with this pattern and will be annotated with corresponding labels.

Property	Pattern	
	Template	Condition
REF	$P_1(a,b)$	$a = b$
	$P_1(a,b)\vee !R_1(c)\vee !R_2(d)$	$a = c, b = d, R_1 = R_2, P_1 \neq R_i$
SYM	$P_1(a,b)\vee !P_2(c,d)$	$a = d, b = c, P_1 = P_2$
	$P_1(a,b)\vee !R_1(c)\vee !R_2(d)$	$a = c, b = d, R_1 = R_2, P_1 \neq R_i$
TRN	$!P_1(a,b)\vee !P_2(c,d)\vee P_3(e,f)$	$b = c, a = e, d = f, P_1 = P_2 = P_3$
KEY	$!P_1(a,b)\vee !P_2(e,f) \vee [c = d]$	$a = e, b = c, d = f, P_1 = P_2$
NoREC	$R_1() \vee \ldots \vee R_n() \vee P_1()$	$P_1 \neq R_i$
	$R_1() \vee \ldots \vee R_n()\vee !P_1()$	$P_1 \neq R_i$
TrRec	$P_1(a,b) \vee T(c,d) \vee P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, T(c,d) = [d = c + x], x \neq 0$
	$P_1(a,b) \vee T(c,d) \vee P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, \forall(c,d) \in T, c \sqsubseteq d$
	$!P_1(a,b) \vee T(c,d) \vee P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, T(c,d) = [d = c + x], x \neq 0$
	$!P_1(a,b) \vee T(c,d) \vee P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, \forall(c,d) \in T, c \sqsubseteq d$
	$P_1(a,b) \vee T(c,d)\vee !P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, T(c,d) = [d = c + x], x \neq 0$
	$P_1(a,b) \vee T(c,d)\vee !P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, \forall(c,d) \in T, c \sqsubseteq d$
	$!P_1(a,b) \vee T(c,d)\vee !P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, T(c,d) = [d = c + x], x \neq 0$
	$!P_1(a,b) \vee T(c,d)\vee !P_2(e,f)$	$b = c, d = f, a = e, P_1 = P_2, \forall(c,d) \in T, c \sqsubseteq d$

Table 8: Sufficient Conditions for Properties. All Patterns for REF, SYM, TRN, and KEY are hard rules.

C.1.3 Static Analysis for Data Partitioning

Statistical inference can often be decomposed as independent subtasks on different portions of the data. Take the examples of classification in Section 2.2 for instance. The inference of the query relation $\texttt{winner}(team)$ is "local" to each *team* constant (Assume $\texttt{label}$ is the evidence relation). In other words, deciding whether one *team* is a winner does not rely on the decision of another team, $team'$, in this classification subtask. Therefore, if there are a total of n teams, we will have an opportunity to solve this subtask using n concurrent threads. Another example is labeling, which is often local to small units of sequences (e.g., sentences).

In FELIX, we borrow ideas from the Datalog literature [35] that uses linear programming to perform static analysis to decompose the data. FELIX adopts the same algorithm of Seib and Larsen [35].

Consider an operator with query relation $\texttt{R}(\bar{x})$. Different instances of $\bar{x}$ may depend on each other during inference. For example, consider the rule

$$\texttt{R}(\bar{x}) <= \texttt{R}(\bar{y}), \texttt{T}(\bar{x}, \bar{y}).$$

Intuitively, all instances of $\bar{x}$ and $\bar{y}$ that appear in the same rule cannot be solved independently since $\texttt{R}(\bar{x})$ and $\texttt{R}(\bar{y})$ are inter-dependent. Such dependency relationships are transitive, and we want to compute them so that data partitioning wouldn't violate them. A straightforward approach is to ground all rules and then perform component detection on the resultant graph. But grounding tends to be very computationally demanding. A cheaper way is static analysis that looks at the rules only. Specifically, one solution is to find a function $f_R(-)$ which has $f_R(\bar{x}) = f_R(\bar{y})$ for all $\bar{x}$ and $\bar{y}$'s that rely on each other. As we rely on static analysis to find f_R, the above condition should hold for all possible database instances.

Assuming each constant is encoded as an integer in FELIX, we may consider functions f_R of the form [35]:

$$f_R(x_1, ..., x_n) = \sum_i \lambda_i x_i \in \mathbb{N},$$

where λ_i are integer constants.

Following [35], FELIX uses linear programming to find λ_i such that $f_R(-)$ satisfy the above constraints. Once we have such a partitioning function over the input, we can process the data in parallel. For example, if we want to run N concurrent threads for $\texttt{R}$, we could assign all data satisfying

$$f_R(x_1, ..., x_n) \mod N = j$$

to the jth thread.

C.2 Operators Implementation

Recall that FELIX selects physical implementations for each logical operator to actually execute them. In this section, we show a handful of physical implementations for these operators. Each of these physical implementations only works for a subset of the operator configurations. For cases not covered by these physical implementations, we can always use TUFFY or Gauss-Seidel-Style implementations [22].

Using Logistic Regression for Classification Operators Consider a Classification operator with a query relation $R(\underline{k}, v)$, where k is the key. Recall that each possible value of k corresponds to an independent classification task. The (ground) rules of this operator are all non-recursive with respect to R, and so can be grouped by value of k. Specifically, for each value pair $\hat{k}$ and $\hat{v}$, define

$$\mathcal{R}_{\hat{k},\hat{v}} = \quad \{r_i | r_i \text{ is violated when } R(\hat{k}, \hat{v}) \text{ is true }\}$$
$$\mathcal{R}_{\hat{k},\perp} = \quad \{r_i | r_i \text{ is violated when } \forall v \ R(\hat{k}, \hat{v}) \text{ is false}\}$$

and

$$W_{\hat{k},x} = \sum_{r_i \in \mathcal{R}_{\hat{k},x}} |w_i|$$

which intuitively summarizes the penalty we have to pay for assigning x for the key $\hat{k}$.

With the above notation, one can check that

$$\Pr[R(\hat{k}, x) \text{ is true}] = \frac{\exp\{-W_{\hat{k},x}\}}{\sum_y \exp\{-W_{\hat{k},y}\}},$$

where both x and y range over the domain of v plus $\perp$, and $R(\hat{k}, \perp)$ means $R(\hat{k}, v)$ is false for all values of v. This is implemented using SQL aggregation in a straightforward manner.

Using Conditional Random Field for Labeling Operators The Labeling operator generalizes the Classification operator by allowing tree-shaped correlations between the individual classification tasks. For simplicity, assume that such tree-shaped correlation is actually a chain. Specifically, suppose the possible values of k are $k_1, \ldots, k_m$. Then in addition to the ground rules as described in the previous paragraph, we also have a set of recursive rules each containing $R(k_i, -)$ and $R(k_{i+1}, -)$ for some $1 \leq i \leq m - 1$. Define

$$\mathcal{R}^B_{k_i,k_{i+1}} = \quad \{r_i | r_i \text{ contains } R(k_i, -) \text{ and } R(k_{i+1}, -)\}$$
$$W^B_{k_i,k_{i+1}}(v_i, v_{i+1}) = \sum_{r_i \in \mathcal{R}^B_{k_i,k_{i+1}}} \text{cost}_{r_i}(\{R(k_i, v_i), R(k_{i+1}, v_{i+1})\}).$$

Then it's easy to show that

$$\Pr[\{R(k_i, v_i), 1 \leq i \leq m\}] \propto \exp\{-\sum_{1 \leq i \leq m} W_{k_i,v_i} - \sum_{1 \leq i \leq m-1} W^B_{k_i,k_{i+1}}(v_i, v_{i+1})\},$$

which is exactly a linear-chain CRF.

Again, FELIX uses SQL to compute the above intermediate statistics, and then resort to the Viterbi algorithm [17] (for MAP inference) or the sum-product algorithm [43] (for marginal inference).

Using Correlation Clustering for Coreference Operators   The Coref operator can be implemented using correlation clustering [4]. We show that the constant-approximation algorithm for correlation clustering carries over to MLNs under some technical conditions. Recall that correlation clustering essentially performs node partitioning based on the edge weights in an undirected graph. We use the following example to illustrate the direct connection between MLN rules and correlation clustering.

Example 1 Consider the following ground rules which are similar to those in Section 2.2:

10	$\texttt{inSameDoc}(P_1, P_2), \texttt{sameString}(P_1, P_2) => \texttt{coRef}(P_1, P_2)$
5	$\texttt{inSameDoc}(P_1, P_2), \texttt{subString}(P_1, P_2) => \texttt{coRef}(P_1, P_2)$
5	$\texttt{inSameDoc}(P_3, P_4), \texttt{subString}(P_3, P_4) => \texttt{coRef}(P_3, P_4)$

Assume $\texttt{coRef}$ is the query relation in this Coreference operator. We can construct the weighted graph as follows. The vertex set is $V = \{P_1, P_2, P_3, P_4\}$. There are two edges with non-zero weight: (P_1, P_2) with weight 15 and (P_3, P_4) with weight 5. Other edges all have weight 0. The following proposition shows that the correlation clustering algorithm solves an equivalent optimization problem as the MAP inference in MLNs.

Proposition C.1. *Let $\Gamma(\bar{x}_i)$ be a part of Γ corresponding to a coref subtask; let G_i be the correlation clustering problem transformed from $\Gamma(\bar{x}_i)$ using the above procedure. Then an optimal solution to G_i is also an optimal solution to $\Gamma(\bar{x}_i)$.*

We implement Arasu et al. [4] for correlation clustering. The theorem below shows that, for a certain family of MLN programs, the algorithm implemented in FELIX actually performs approximate MLN inference.

Theorem C.1. *Let $\Gamma(\bar{x}_i)$ be a coref subtask with rules generating a complete graph where each edge has a weight of either $\pm\infty$ or w s.t. $m \leq |w| \leq M$ for some $m, M > 0$. Then the correlation clustering algorithm running on $\Gamma(\bar{x}_i)$ is a $\frac{3M}{m}$-approximation algorithm in terms of the log-likelihood of the output world.*

Proof. In Arasu et al. [4], it was shown that for the case $m = M$, their algorithm achieves an approximation ratio of 3. If we run the same algorithm, then in expectation the output violates no more than $3\mathbf{OPT}$ edges, where $\mathbf{OPT}$ is the number of violated edges in the optimal partition. Now with weighted edges, the optimal cost is at least $m\mathbf{OPT}$, and the expected cost of the algorithm output is at most $3M\mathbf{OPT}$. Thus, the same algorithm achieves $\frac{3M}{m}$ approximation. $\square$

C.3   Cost Model for Physical Optimization

The cost model in Section 4.2 requires estimation of the individual terms in ExecCost. There are three components: (1) the materialization cost of each eager query, (2) the cost of lazily evaluating the query in terms of the materialized views, and (3) the number of times that the query will be executed (t). We consider them in turn.

Computing (1), the subquery materialization cost $\text{Mat}(Q_i)$, is straightforward by using PostgreSQL's EXPLAIN feature. As is common for many RDBMSs, the unit of PostgreSQL's query evaluation cost is not time, but instead an internal unit (roughly proportional to the cost of 1 I/O). FELIX performs all calculations in this unit.

Computing (2), the cost of a single incremental evaluation, is more involved: we do not have Q_i actually materialized (and with indexes built), so we cannot directly measure $\text{Inc}_Q(Q')$ using PostgreSQL. For simplicity, consider a two-way decomposition of Q into Q_1 and Q_2. We consider two cases: (a) when Q_2 is estimated to be larger than PostgreSQL assigned buffer, and (b) when Q_2 is smaller (i.e. can fit in available memory).

To perform this estimation in case (a), FELIX makes a simplifying assumption that the Q_i are joined together using index-nested loop join (we will build the index when we actually materialize the tables). Exploring clustering opportunities for Q_i is future work.

Then, we force the RDBMS to estimate the detailed costs of the plan $\mathcal{P} : \sigma_{\bar{x}'=\bar{a}}(Q_1) \bowtie \sigma_{\bar{x}'=\bar{a}}(Q_2)$, where Q_1 and Q_2 are views, $\bar{x}' = \bar{a}$ is an assignment to the bound variables $\bar{x}' \equiv \bar{x}^{\mathsf{b}}$ in $\bar{x}$. From the detailed cost estimation, we extract the following quantities: (1) n_i: be the number of tuples from subquery $\sigma_{\bar{x}}(Q_i)$; (2) n:

the number of tuples generated by $\mathcal{P}$. We also estimate the cost α (in PostgreSQL's unit) of each I/O by asking PostgreSQL to estimate the cost of selections on some existing tables.

Denote by $c' = \text{Inc}_Q(Q')$ the cost (in PostgreSQL unit) of executing $\sigma_{\bar{x}'=\bar{a}}(R_1) \bowtie \sigma_{\bar{x}'=\bar{a}}(R_2)$, where R_i is the materialized table of Q_i with proper indexes built. Without loss of generality, assume $n_1 < n_2$ and that n_1 is small enough so that $\bowtie$ in the above query is executed using nested loop join. On average, for each of the estimated n_1 tuples in $\sigma_{\bar{x}}(R_1)$, there is one index access to R_2, and $\lceil \frac{n}{n_1} \rceil$ tuples in $\sigma_{\bar{x}}(R_2)$ that can be joined; assume each of the $\lceil \frac{n}{n_1} \rceil$ tuples from R_2 requires one disk page I/O. Thus, there are $n_1 \lceil \frac{n}{n_1} \rceil$ disk accesses to retrieve the tuples from R_2, and

$$c' = \alpha n_1 \left[\lceil \frac{n}{n_1} \rceil + \log |Q_2| \right] \tag{11}$$

where we use $\log |Q_2|$ as the cost of one index access to R_2 (height of a B-tree). Now both $c' = \text{Inc}_Q(Q')$ and $\text{Mat}(Q_i)$ are in the unit of PostgreSQL cost, we can sum them together, and compare with the estimation on other materialization plans.

In case (b), when Q_2 can fit in memory, we found that the above estimation tends to be too conservative – many accesses to Q_2 are cache hits whereas the model above still counts the accesses into disk I/O. To compensate for this difference, we multiply c' (derived above) with a fudge factor $\beta < 1$. Intuitively, we choose β as the ratio of accessing a page in main memory versus accessing a page on disk. We empirically determine β.

Component (3) is the factor t, which is dependent on the statistical operator. However, we can often derive an estimation method from the algorithm inside the operator. For example, for the algorithm in [4], the number of requests to an input data movement operator can be estimated by the total number of mentions (using COUNT) divided by the expected average node degree.

D   Additional Experiments

D.1   Additional Experiments of High-level Quality and Performance

We describe the detailed methodology in our experiments on the Enron-R, DBLife, and NFL datasets.

Enron-R   We compare the performance and quality of FELIX with SYSTEMT on Enron-R data set. SYSTEMT uses dictionaries for person name extraction, and regular expressions for phone number extraction. To extract person-phone relationships, SYSTEMT uses a fixed window size to identify person-phone co-occurrences, and translate those co-occurrences directly into person-phone relationships. We vary this window size to produce a precision-recall curve of SYSTEMT, as shown in Figure 5(a).

We next write an MLN program to replace SYSTEMT's relation extraction part (using the same entity extraction results). Instead of fixed window sizes, this program uses MLN rule weights to encode the strength of co-occurrence and thereby confidence in person-phone relationships. In addition, we write soft constraints such as "*a phone number cannot be associated with too many persons.*" We add in a set of coreference rules to perform person coref. We run ALCHEMY, TUFFY and FELIX on this program.

We plot the quality results in Figure 5(a). We see that FELIX and TUFFY clearly dominates SYSTEMT. For example, at the same recall, FELIX achieves about twice as high precision as SYSTEMT. Another observation is FELIX achieves similar quality with TUFFY. This is not surprising because FELIX's operator-based framework still follows MLN's semantic. Although TUFFY runs on Enron-R and achieves significant improvement compared with SYSTEMT, on all other datasets TUFFY crashes (including the full Enron dataset).

DBLife   We compare the performance and quality of FELIX with CIMPLE. CIMPLE identifies person and organization mentions using dictionaries with regular expression variations (e.g., abbreviations, titles). In case of an ambiguous mention such as "J. Smith", CIMPLE binds it to an arbitrary name in its dictionary that is

compatible (e.g., "John Smith"). CIMPLE then uses a proximity-based formula to translate person-organization co-occurrences into ranked affiliation tuples.

Felix works as follows. We first extract entities from the corpus. We perform part-of-speech tagging [34] on the raw text, and then identify possible person/organization names using simple heuristics (e.g., common person name dictionaries and keywords such as "University"). To handle noise in the entity extraction results, our MLN program performs both affiliation extraction and coref resolution using ideas similar to Figure 2. We run FELIX, TUFFY, and ALCHEMY on this program. TUFFY and ALCHEMY crashed without producing any results.

We plot the precision-recall curves of CIMPLE and FELIX by varying the k in "*top-k affiliations of each person*" (Figure 5(b)). For any given precision, FELIX has substantially higher recall than CIMPLE. This suggests that statistical approaches may help deterministic rule-based systems: at the same precision, the recall of FELIX is 2-3 times as high as CIMPLE.

NFL On the NFL dataset, we extract winner-loser pairs. There are 1,100 sports news articles in the corpus. We obtain ground truth of game results from the web. As the baseline solution, we use 610 of the articles together with ground truth to train a CRF model that tags each token in the text as either WINNER, LOSER, or OTHER. We then apply this CRF model on the remaining 500 articles to generate probabilistic tagging of the tokens. Those 500 articles report on a different season of NFL games than the training articles, and we have ground truth on game results (in the form of winner-loser-date triples). We take the publication dates of the articles and align them to game dates. For each sentence containing a WINNER token with probability p and a LOSER token with probability q, if both tokens can be resolved to NFL team names, we emit a pair of these teams with score $(p+q)/2$. This generates a ranked list of winner-loser-date triples. We plot the precision-recall curve of this result as **CRF** in Figure 5(c).

The MLN program on NFL consists of two parts. The first part contains MLN rules encoding the CRF model for winner/loser team mention extraction. The second part is adapted from the rules developed by a research team in the Machine Reading project. Those rules model simple domain knowledge such as "a winner cannot be a loser on the same day" and "a team cannot win twice on the same day." We also add in coreference of the team mentions. We run this program on FELIX, TUFFY and ALCHEMY. TUFFY fails to generate any results in 6 hours when attempting to generate 10^8 tuples during grounding. ALCHEMY does not generate any results in 6 hours. In contrast, FELIX runs smoothly and produces result quality that is significantly superior than CRF. This improvement demonstrates the desirability of statistical inference. Moreover, to cope with complex programs, the proposed operator-based approach (as done by FELIX) outperforms a monolithic approach (as done by TUFFY).

	Coref	Labeling	Classification	MLN Inference
Enron-R	1/1	0/0	0/0	1/1
DBLife	2/2	0/0	1/1	0/0
NFL	1/1	1/1	0/0	1/1
Program1	0/0	1/1	0/0	0/0
Program2	0/0	0/0	37/37	0/0
Program3	0/0	0/1	0/0	1/1

Table 9: Specialized Operators Discovered by Felix's Compiler

D.2   Coverage of the Compiler

Since discovering subtasks as operators is crucial to FELIX's scalability, in this section we test FELIX's compiler. We first evaluate the heuristics we are using for discovering statistical operators given an MLN program. We then evaluate the performance of the $\Pi_2 P$-complete algorithm to discovering REF and SYM in non-recursive programs.
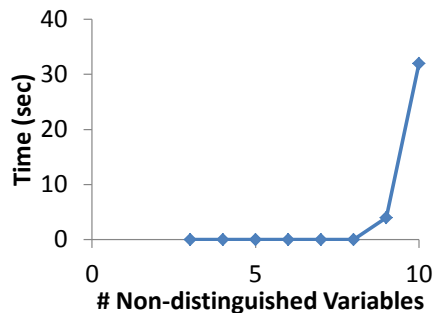
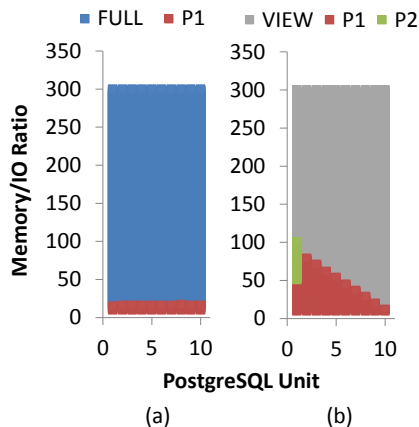Figure 7: Performance of Π_2P-complete Algorithms for Non-recursive Programs

Figure 8: Plan diagram of FELIX's Cost Optimizer

Using Heuristics for Arbitrary MLN Programs While FELIX's compiler can discover all Coref, Labeling, and Classification operators in all programs used in our experiments, we are also interested in how many operators FELIX can discover from other programs. To test this, we download the programs that are available on ALCHEMY's Web site [23] and manually label operators in these programs. We label a set of rules as an operator if this set of rules follows our definition of statistical operator in Section 3.1.

We then run FELIX's compiler on these programs and compare the logical plans produced by FELIX with our manual labels. We list all programs with manually labeled operators in Table 9. The x/y in each cell of Table 9 means that, among y manually labeled operators, Felix's compiler discovers x of them.

We can see from Table 9 that FELIX's compiler works well for the programs used in our experiment. Also, FELIX works well on discovering classification and labeling operators in ALCHEMY's programs. This implies the set of heuristic rules we are using, although not complete, indeed encodes some popular patterns users may use in real world applications. Although some of ALCHEMY's programs encode coreference resolution tasks, none of them were labeled as coreference operator. This is because none of these programs explicitly declares the symmetric constraints as hard rules. Therefore, the set of possible worlds decided by the MLN program is different from those decided by the typical "partitioning"-based semantics of coreference operators. How to detect and efficiently implement these "soft-coref" is an interesting topic for future work.

Performance of Π_2P-complete Algorithm for Non-recursive Programs In Section 4.1 and Section C.1.1 we show that there are Π_2P-complete algorithms for annotating REF and SYM properties. FELIX implements them. As the intractability is actually inherent in the number of non-distinguished variables, which is usually small, we are interested in understanding the performance of these algorithms.
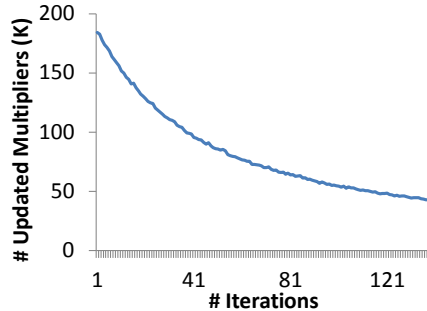
---

[23]http://alchemy.cs.washington.edu/mlns/

Figure 9: Convergence of Dual Decomposition

We start from one of the longest rules found in ALCHEMY's Web site which can be annotated as SYM. This rule has 3 non-distinguished variables. We then add more non-distinguished variables and plot the time used for each setting (Figure 7). We can see that FELIX uses less than 1 second to annotate the original rule, but exponentially more time when the number of non-distinguished variables grows to 10. This is not surprising due to the exponential complexity of this algorithm. Another interesting conclusion we can draw from Figure 7 is that, as long as the number of non-distinguished variables is less than 10 (which is usually the case in our programs), FELIX performs reasonably efficiently.

D.3 Stability of Cost Estimator

In our previous experiments we show that the plan generated by FELIX's cost optimizer contributes to the scalability of FELIX. As the optimizer needs to estimate several parameters before performing any predictions, we are interested in the sensitivity of our current optimizer to the estimation errors of these parameters.

The only two parameters used in FELIX's optimizer are 1) the cost (in PostgreSQL's unit) of fetching one page from the disk and 2) the ratio of the speed between fetching one page from the memory and fetching one page from the disk. We test all combined settings of these two parameters ($\pm 100\%$ of the estimated value) and draw the plan diagram of two queries in Figure 8. We represent different execution plans with different colors. For each point (x, y) in the plan diagram, the color of that point represents which execution plan the compiler chooses if the PostgreSQL's unit equals x and memory/IO ratio equals y.

For those queries not shown in Figure 8, FELIX produces the same plan for each tested parameter combination. For queries shown in Figure 8, we can see FELIX is robust for parameter mis-estimation. Actually, all the plans shown in Figure 8 are close to optimal, which implies that in our experiments FELIX's cost optimizer avoids the selection of "extremely bad" plans even under serious mis-estimation of parameters.

D.4 Convergence of Dual Decomposition

FELIX implements an iterative approach for dual decomposition. One immediate question is *how many iterations do we need before the algorithm converges?*.

To gain some intuitions, we run FELIX on the DBLife [24] data set for a relative long time and record the number of updated Lagrangian multipliers of each iteration. We use constant step size $\lambda = 0.9$. As shown in Figure 9, even after more than 130 iterations, the Lagrangian multipliers are still under heavy updates. However, on the ENRON-R dataset, we observed that the whole process converges after the first several iterations! This implies that the convergence of our operator-based framework depends on the underlying MLN program and the size of the input data. It is interesting to see how different techniques on dual decomposition and gradient methods can alleviate this convergence issue, which we leave as future work.

Fortunately, we empirically find that in all of our experiments, taking the result from the first several iterations is often a reasonable trade-off between time and quality – all P/R curves in the previous experiments

---

[24]Similar phenomena occur in the NFL dataset as well.

are generated by taking the last iteration within 3000 seconds and we already get significant improvements compared to baseline solutions. In FELIX, to allow users to directly trade-off between quality and performance, we provide two modes: 1) Only run the first iteration and flush the result immediately; and 2) Run the number of iterations specified by the user. It is an interesting direction to explore the possibility of automatically selecting parameters for dual decomposition.