Querying Probabilistic Information Extraction

Daisy Zhe Wang*, Michael J. Franklin*, Minos Garofalakis†, and Joseph M. Hellerstein*
*University of California, Berkeley and †Technical University of Crete

ABSTRACT

Recently, there has been increasing interest in extending relational query processing to include data obtained from unstructured sources. A common approach is to use stand-alone Information Extraction (IE) techniques to identify and label entities within blocks of text; the resulting entities are then imported into a standard database and processed using relational queries. This two-part approach, however, suffers from two main drawbacks. First, IE is inherently probabilistic, but traditional query processing does not properly handle probabilistic data, resulting in reduced answer quality. Second, performance inefficiencies arise due to the separation of IE from query processing. In this paper, we address these two problems by building on an in-database implementation of a leading IE model—Conditional Random Fields using the Viterbi inference algorithm. We develop two different query approaches on top of this implementation. The first uses deterministic queries over maximum-likelihood extractions, with optimizations to push the relational operators into the Viterbi algorithm. The second extends the Viterbi algorithm to produce a set of possible extraction "worlds", from which we compute top-k probabilistic query answers. We describe these approaches and explore the trade-offs of efficiency and effectiveness between them using two datasets.

1 Introduction

The field of database management has traditionally focused on structured data, providing little or no help for the significantly larger amounts of the world's data that is unstructured. With the rise of text-based applications on the web and elsewhere, there has been significant progress in information extraction (IE) techniques, which parse text and extract structured objects that can be integrated into databases for querying.

In the database community, work on IE has centered on two major architectural themes. First, there has been interest in the design of declarative languages and systems to easily specify, optimize and execute IE tasks [1, 2, 3]. Second, IE has been a primary motivating application for the groundswell of work on Probabilistic Database Systems (PDBS) [4, 5, 6, 7, 8, 9], which can model the uncertainty inherent in IE outputs, and enable users to write declarative queries that deal with such uncertainty.

Given this context, our goal is to merge these two ideas into a single architecture, and build the first unified database system that provides a query-oriented language for specifying, optimizing, and executing IE tasks, and that supports a principled probabilistic framework for querying the outputs of those tasks.

In previous work, we described a DBMS-based implementation of Conditional Random Fields (CRF)—a leading probabilistic IE model, and Viterbi—the maximum-likelihood (ML) inference algorithm over CRF [10]. Taking only the ML extraction results and ignoring their associated probabilities, enables relational queries to be performed in a straightforward manner. This approach, however, suffers from two important limitations: (1) as shown in [11], restricting the queries to the ML extractions can result in incorrect answers, and (2) since the IE inference is computed separately from the queries over the extracted data, important optimization opportunities are lost.

In this paper, we aim to achieve a deeper integration of IE and inference with relational queries to improve both run-time efficiency and answer quality. We study two approaches.

In the first, we consider deterministic Select-Project-Join (SPJ) queries over the ML results of Viterbi-based IE. For this case, we develop query optimizations that reduce work by carefully pushing relational operators into the Viterbi algorithm. Compared to previous approaches that treated the IE and query phases separately, these optimizations can significantly improve efficiency.

The key problem with querying the ML extractions stems from inaccuracies in IE models: even high-quality ML extractions contain errors, which can be exacerbated in SPJ queries over such extractions. We illustrate this point with an example scenario.

EXAMPLE 1. *Kristjansson et al. [12] used CRF-based* IE *to extract* Contact *information from the signature blocks in the Enron email corpus [13]. One such block begins with the text:*

Michelle L. Simpkins
Winstead Sechrest & Minick P.C.
100 Congress Avenue, Suite 800...

One of the fields we wish to extract from these email blocks is companyname, which in this case should be "Winstead Sechrest & Minick P.C.". Unfortunately, the ML extraction from the CRF model assigns the value NULL to companyname. This IE *error shows up more explicitly in the following relational queries:*

Query 1. Find contacts with companyname containing *"Winstead"*
```
SELECT *
FROM Contacts
WHERE companyname LIKE '%Winstead%'
```

Query 2. Find all contact pairs with the same companyname
```
SELECT *
FROM Contacts C1, Contacts C2
WHERE C1.companyname = C2.companyname
```

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

The first query provides an empty result, since "Winstead Sechrest & Minick P.C." is not identified as a companyname. The second query is perhaps more vexing and counterintuitive: the erroneous NULL parse for Michelle Simpkins' companyname means that she does not even match herself, *hence, she is again absent from the output. In general, due to the inherent uncertainty in* IE, *the answers to queries over ML extractions can contain false negatives and/or false positives.* □

To address this answer quality problem, we develop an alternative to the ML-based approach using Probabilistic Database techniques, where *probabilistic* SPJ queries are computed over the set of possible "worlds" (PWs) induced from the CRF-based distribution.

We show that, by allowing us to look beyond ML extractions, this probabilistic database approach can significantly reduce false negatives (by as much as 80% in our experiments) with only marginal increase in false positives. As expected, the improvement in answer quality comes at a performance cost due to the consideration of the full CRF distribution instead of only the ML extraction. For probabilistic selection, we show how to reduce this overhead to a small fixed-cost per document over the optimized ML-based queries. For probabilistic join, however, the performance penalty can be much higher. Thus, the two approaches we develop, while both improvements over the state-of-the-art, provide a design space where performance or answer quality can be emphasized based on the requirements of the application.

Our key contributions in developing an integrated approach to querying and IE can be summarized as follows:

- We present novel optimizations for *SPJ queries over the maximum-likelihood world* of the CRF distribution that improve performance through the effective integration of relational operators and Viterbi-style inference;
- We propose new algorithms, that substantially extend vanilla-Viterbi inference to enable effective *probabilistic SPJ queries over the full CRF distribution*, and, thus, improved answer quality by considering the full set of possible extraction "worlds";
- We evaluate our proposed approaches and algorithms, demonstrating the scalability of our solutions and exploring the trade-off between efficiency and answer quality using data collections from the Enron and DBLP datasets.

2 Background

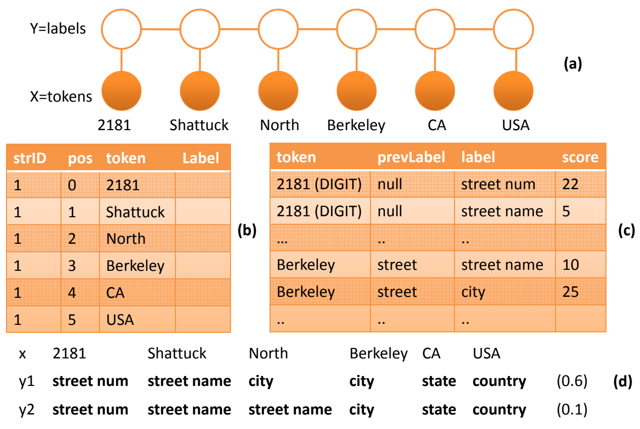
This section covers the concept of a probabilistic database and the different types of inference operations over a CRF model, particularly in the context of information extraction.

2.1 Probabilistic Databases

A *probabilistic database* $\mathcal{DB}^p$ consists of two key components: (1) a collection of incomplete relations $\mathcal{R}$ with missing or uncertain data, and (2) a probability distribution F on all possible database instances, which we call *possible worlds*, and denote by $pwd(D^p)$. The attributes of an incomplete relation $R \in \mathcal{R}$ include a subset that are *probabilistic attributes* $\mathcal{A}^p$, whose values may be present, missing or uncertain. Each possible database instance is a possible completion of the missing and uncertain data in $\mathcal{R}$.

2.2 Conditional Random Fields (CRF)

The linear-chain CRF [14, 15], similar to the Hidden Markov Model, is a leading probabilistic model for solving IE tasks. In the context of IE, a CRF model encodes the probability distribution over a set of *label* random variables (RVs) $\mathbf{Y}$, given the value of a set of *token* RVs $\mathbf{X}$. We denote an assignment to $\mathbf{X}$ by $\mathbf{x}$ and to $\mathbf{Y}$ by $\mathbf{y}$. In a linear-chain CRF model, label y_i is correlated only with label y_{i-1} and token x_i. Such correlations are represented by the feature functions $\{f_k(y_i, y_{i-1}, x_i)\}_{k=1}^{K}$.



strID	pos	token	Label
1	0	2181	
1	1	Shattuck	
1	2	North	
1	3	Berkeley	
1	4	CA	
1	5	USA	

token	prevLabel	label	score
2181 (DIGIT)	null	street num	22
2181 (DIGIT)	null	street name	5
...	..	..	
Berkeley	street	street name	10
Berkeley	street	city	25
..	..	..	

x	2181	Shattuck	North	Berkeley	CA	USA	
y1	street num	street name	city	city	state	country	(0.6)
y2	street num	street name	street name	city	state	country	(0.1)

Figure 1: (a) Example CRF model; (b) Example TOKENTBL table; (c) Example FACTORTBL table; (d) Two possible segmentations $\mathbf{y}_1, \mathbf{y}_2$.

EXAMPLE 2. *Figure 1(a) shows an example CRF model over an address string* $\mathbf{x}$ *'2181 Shattuck North Berkeley CA USA'. Observed (known) variables are shaded nodes in the graph. Hidden (unknown) variables are unshaded. Edges in the graph denote statistical correlations. The possible labels are* $Y = \{$*apt.num, street-num, streetname, city, state, country*$\}$. *Two possible feature functions of this CRF are:*

$$\begin{array}{rcl} f_1(y_i, y_{i-1}, x_i) & = & [x_i \textit{ appears in a city list}] \cdot [y_i = \textit{city}] \\ f_2(y_i, y_{i-1}, x_i) & = & [x_i \textit{ is an integer}] \cdot [y_i = \textit{apt.num}] \\ & & \cdot [y_{i-1} = \textit{streetname}] \end{array}$$

A segmentation $\mathbf{y} = \{y_1, ..., y_T\}$ *is one possible way to tag each token in* $\mathbf{x}$ *with one of the labels in* Y. *Figure 1(d) shows two possible segmentations of* $\mathbf{x}$ *and their probabilities.* □

DEFINITION 2.1. *Let* $\{f_k(y_i, y_{i-1}, x_i)\}_{k=1}^{K}$ *be a set of real-valued feature functions, and* $\Lambda = \{\lambda_k\} \in R^K$ *be a vector of real-valued parameters, a CRF model defines the probabilistic distribution of segmentations* $\mathbf{y}$ *given a specific token sequence* $\mathbf{x}$:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{1}{Z(\mathbf{x})} exp\{\sum_{i=1}^{T}\sum_{k=1}^{K} \lambda_k f_k(y_i, y_{i-1}, x_i)\}, \tag{1}$$

where $Z(\mathbf{x})$ *is a standard normalization function that guarantees probability values between* 0 *and* 1. □

2.3 Inference Queries over a CRF Model

There are three types of inference queries over the CRF model [15]. **Top-k Inference**: The top-k inference computes the segmentations with the top-k highest probabilities given a token sequence $\mathbf{x}$ from a text-string d. The Viterbi dynamic programming algorithm [16] is the key algorithmic technique for CRF top-k inference.

The Viterbi algorithm computes a two dimensional V matrix, where each cell $V(i, y)$ stores a ranked list of *entries* $e = \{score, prev(label, idx)\}$ ordered by *score*. Each entry contains (1) the *score* of a top-k (partial) segmentation ending at position i with label y; and, (2) a pointer to the previous entry *prev* on the path that led to top-k *score*'s in $V(i, y)$. The pointer $e.prev$ consists of the label *label* and the list index *idx* of the previous entry on the path to e. Based on Equation (1), the recurrence to compute the ML (top-1) segmentation is as follows:

$$V(i, y) = \begin{cases} \max_{y'}(V(i-1, y') \\ \quad + \sum_{k=1}^{K} \lambda_k f_k(y, y', x_i)), & \text{if} \quad i \geqslant 0 \\ 0, & \text{if} \quad i = -1. \end{cases} \tag{2}$$

The ML segmentation y^*, backtracked from the maximum entry in $V(T, y_T)$ (where T is the length of the token sequence $\mathbf{x}$) through *prev* pointers, is shown in bold arrows in Figure 2(a). The complexity of the Viterbi algorithm is $O(T \cdot |Y|^2)$, where $|Y|$ is the number of possible labels.

Constrained Top-k Inference: Constrained top-k inference [12] is a special case of top-k inference. It is used when a subset of the token labels has been provided (e.g., via a user interface). Let $\mathbf{s}$ be the evidence vector $\{s_1, ..., s_T\}$, where s_i is either $NULL$ (i.e., no evidence) or the evidence label for y_i. Constrained top-k inference can be computed by a variant of the Viterbi algorithm which restricts the chosen labels $\mathbf{y}$ to conform with the evidence $\mathbf{s}$.

Marginal Inference: Marginal inference computes a marginal probability $p(y_t, y_{t+1}, ..., y_{t+k}|\mathbf{x})$ over a single label or a sub-sequence of labels [15]. The backward-forward algorithm, a variation of the Viterbi algorithm, is used for such marginal inference tasks.

3 Setup

This section describes the setup of the system. Unstructured text is represented by a set of documents or text-strings $\mathcal{D}$, and each document $d \in \mathcal{D}$ is represented by a set of token records in TOKENTBL. The CRF-based distribution over documents $\mathcal{D}$ is stored in FACTORTBL. The TOKENTBL and FACTORTBL are pre-materialized. Based on TOKENTBL and FACTORTBL, each document is parsed into one probabilistic record in an *entity table*. Two families of queries over the probabilistic entity tables are explored.

Token Table: The token table TOKENTBL is an incomplete relation R in $\mathcal{DB}^p$, which stores text-strings as relations in a database, in a manner akin to the inverted files commonly used in information retrieval. The TOKENTBL contains one probabilistic attribute—label^p, whose values need to be inferred. As shown in Figure 1(b), each tuple in TOKENTBL records a unique occurrence of a token, which is identified by the text-string ID (strID) and the position (pos) the token is taken from.

TOKENTBL (strID, pos, token, label^p)

Factor Table: The probability distribution F over all possible "worlds" of TOKENTBL can be computed from the FACTORTBL. The FACTORTBL is a materialization of the *factor tables* in the CRF model for all the tokens in the corpus $\mathcal{D}$. A factor table $\phi[y_i, y_{i-1} \mid x_i]$, which represents the correlation between x_i, y_i, and y_{i-1}, is computed by the weighted sum of a set of feature functions in the CRF model: $\phi[y_i, y_{i-1} \mid x_i] = \sum_{k=1}^{K} \lambda_k f_k(y_i, y_{i-1}, x_i)$.

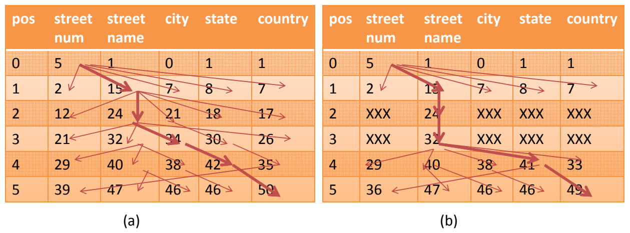
An example of the FACTORTBL is shown in Figure 1(c). A more efficient representation for FACTORTBL is to store the factor table for each token x_i as an array data type, where the array contains a set of scores ordered by {prevLabel, label} [10].

FACTORTBL (token, score ARRAY[])

Entity Table: An entity table contains a set of probabilistic attributes, one for each label in Y. Each tuple in the entity table has an independent distribution, defined by the CRF model over the corresponding text-string d. Figure 3(a) shows the maximum-likelihood (ML) view of the entity table with three address strings.

The entity table is defined and generated by a *pivot operation* over the possible labelings in the TOKENTBL and their distribution. For each possible labeling of a text-string, the pivot operation generates one possible world of the corresponding record in the entity table. For example, the labeling corresponding to segmentation $\mathbf{y}_1$ in Figure 1(d) generates the first tuple in the entity table in Figure 3(a).

Two Families of SPJ Queries: There are two families of queries we consider computing on the extracted data in the entity tables. Given that the ML view of an entity table can be defined as:



(a)

pos	street num	street name	city	state	country
0	5	1	0	1	1
1	2	15	7	8	7
2	12	24	21	18	17
3	21	32	31	30	26
4	29	40	38	42	35
5	39	47	46	46	50

(b)

pos	street num	street name	city	state	country
0	5	1	0	1	1
1	2	15	7	8	7
2	XXX	24	XXX	XXX	XXX
3	XXX	32	XXX	XXX	XXX
4	29	40	38	41	33
5	36	47	46	46	49

Figure 2: Illustration of the computation of V matrix in the following algorithms: (a) Viterbi; (b) ConstrainedViterbi.

```
CREATE VIEW entityTbl1-ML as
  SELECT *, rank() OVER (ORDER BY prob(*) DESC) r
  FROM   entityTbl1
  WHERE  r = 1;
```

one family is the set of deterministic SPJ queries over the ML views of the entity tables. The other family computes the top-k results of probabilistic SPJ queries over entity tables using the set of possible worlds (i.e., segmentations) induced by the CRF distribution. Optionally, a prescribed threshold can be applied to the result probabilities. The queries in this family have the following general form, where `SQLQuery` is a view using a standard SPJ query:

```
SELECT *, rank() OVER (ORDER BY prob(*) DESC) r
FROM   SQLQuery
WHERE  r <= k [ AND prob(*) > threshold ]
```

4 Querying the ML World

In this section, we focus on the deterministic SPJ queries over the ML views of the entity tables. The naive way to compute this type of queries is to first compute the ML extraction for each document, then execute the SPJ queries over the result. We show that for selection and join, conditions can be pushed into the Viterbi inference process to achieve significant speed-up. This performance improvement demonstrates the benefit of a deep integration of IE and relational operators.

4.1 Optimized Selection over ML World

An example of a selection query over the ML view of an entity table is to find all the *Address* tuples whose *streetname* contains 'Sacramento' in the ML extraction:

```
SELECT *   FROM Address-ML
WHERE streetname like '%Sacramento%'
```

The selection condition over the ML view of the *Address* table is rewritten into two selection conditions over the underlying TOKENTBL and FACTORTBL: (1) test if the text-string d contains the token sequence in the selection condition $\mathtt{x}_{cond}$ (e.g., *'Sacramento'*); (2) test if the position(s) where $\mathtt{x}_{cond}$ appears in d are assigned the label in the selection condition y_{cond} (e.g., *streetname*). The first condition can be simply pushed down over the TOKENTBL, only picking the documents that contain the token sequence $\mathtt{x}_{cond}$. The second condition can be pushed into the Viterbi inference computation of a particular text-string d with condition $\{i, len, y'\}$ ensuring that the label at positions from i to $(i+len-1)$ is y'. In general, condition (2) may span multiple tokens. Next, we describe SELVITERBI, a novel variant of the Viterbi algorithm that pushes the condition $\{i, len, y'\}$ into the Viterbi computation.

Recall the Viterbi algorithm described in Section 2.3 which computes a V matrix as in Figure 2(a): The condition $\{i, len, y'\}$ is satisfied if and only if the ML segmentation path backtracks to all the entries in $V(i, y')$ .. $V((i + len - 1), y')$. The intuition behind the SELVITERBI algorithm is as follows. Given condition $\{i, 1, y'\}$ and a text-string d, if none of the top-1 (partial) segmentations ending at position j where $T \geqslant j > i$ backtrack to cell

strID	apt. num^p	street num^p	street name^p	city^p	state^p	country^p
1	null	2181	Shattuck	North Berkeley	CA	USA
2	12B	331	Fillmore St.	Seattle	WA	USA
3	224B	null	Ford South St.	Louis	MO	USA

(a)

strID	company name^p	city^p	state^p
1	Google	Mountain View	CA
2	Yahoo!	Santa Clara	CA
3	Microsoft	Seattle	WA

(b)

Figure 3: Examples of the ML views of entity tables: (a) address strings (b) company location strings.

$V(i, y')$, then we can stop the inference and conclude that this text-string does not satisfy the condition. We call such a position j a *pruning position*, and we would like to detect a pruning position as early as possible to stop the inference computation. Similar intuition holds for the general case where $len > 1$. Our SELVITERBI algorithm follows a Viterbi dynamic program which calls a User Defined Function (UDF) to compute the top (partial) segmentations in $V(i, y_i)$ with additional logic to check if i is a smallest pruning position (thus stopping the recurrence). Variants of SELVITERBI can be employed for increased efficiency based on criteria such as the position and selectivity of a selection condition (which can be incorporated into a cost-based optimizer). A more detailed discussion on SELVITERBI() can be found in Appendix A.

EXAMPLE 3. *Using the example in Figure 2(a), suppose the condition $\{1, 1,$streetnumber$\}$ is generated from a selection condition for a specific text-string d: return text-string d with streetnumber as the label of token $\#2$ (counting from 0) in the ML segmentation. In the second recursive step, only the partial segmentation in $V(1,$streetnumber$)$ satisfies the condition. In the third recursive step, because no partial segmentations in $V(2, y), y \in Y$ come from cell $V(1,$streetnumber$)$ as shown in Figure 2(a), $j = 2$ is the smallest pruning position. Thus, we stop the dynamic programming algorithm and conclude that d does not satisfy the condition $\{1, 1,$streetnumber$\}$.*

4.2 Optimized Join over ML World

An example of a join query over the ML views of two entity tables is to find all (*Address*, *Company*) entity pairs (as in Figure 3) with the same *city* value.

```
SELECT *   FROM Address-ML, Company-ML
WHERE Address-ML.city = Company-ML.city
```

The naive join algorithm over the ML world is: first, compute the Viterbi inference on the two input document sets independently; second, perform the pivot operations to compute the ML views of the *Address* and *Company* entity tables – *Address-ML* and *Company-ML*; finally, compute the deterministic join over the *city* attribute.

The intuition behind the optimization is that we do not need to compute the Viterbi inference over a text-string in the outer document set if it contains none of the join-key values computed from the inference over the inner document set. The optimized algorithm for join-over-ML follows these steps: (1) compute the Viterbi inference over the smaller of the two input document sets, we call this the inner set and the other the outer set; (2) build a hash-table of the join-attribute values computed from the ML extraction of the inner set; (3) perform Viterbi inference only on the documents in the outer set that contain at least one of the hashed values; and (4) perform the pivot operation to compute the ML views of the entity tables and compute the deterministic join over them. This optimization reduces the number of documents on which the Viterbi inference is performed, which improves the performance of join-over-ML queries (by more than a factor of 5 in our experiments).

5 Querying the Full Distribution

In this section, we discuss novel probabilistic query processing strategies to compute the top-k results of SPJ queries over the possible worlds of CRF-based entity tables. In contrast to the techniques in Section 4, the algorithms here do not focus on the single ML segmentation, but rather compute results from the CRF-induced distribution of possible segmentation worlds. As a first step, we present an *incremental* Viterbi algorithm that supports efficient sorted access to the possible segmentations of a document by descending probabilities. This ordering is critical for efficiently computing the top-k results of SPJ queries, especially probabilistic joins.

Due to space constraints, detailed descriptions (e.g., at the pseudo-code level) and analyses for the algorithms in this section can be found in Appendix A.

5.1 Incremental Viterbi Inference

The conventional Viterbi top-k inference algorithm is a straightforward adaptation of the dynamic programming recurrence in Equation (2). The idea is to maintain, in each $V(i, y)$ cell, a list of the top-k partial segmentations ending at position i with label y. This, of course, assumes that the "depth" parameter k is known a priori.

We present a variant of the Viterbi algorithm that can *incrementally* (through a GET-NEXT-SEG() call) compute the next highest-probability segmentation of a document. Our algorithm is similar in spirit to the theoretical framework of Huang et al. [17] for finding k-best trees in the context of hypergraph traversals.

Recall the V matrix computed by the Viterbi algorithm, where each cell $V(i, y)$ contains a list of top-k entries $e = \{score, prev(label, idx)\}$ ordered by $score$. The key idea in our incremental algorithm is to enumerate this list lazily: only on the path of the last extracted segmentation.

Since all the $V(T)$ (T is the length of the document) entries in already-extracted (i.e., "consumed") segmentation paths cannot be used to compute the next highest-probability segmentation, they recursively trigger (through a GET-NEXT() call) the computation for the next "unconsumed" entry using Equation (2) from their predecessor V cells. The key observation here is that these recursive GET-NEXT() calls can only occur *along the path of the last extracted segmentation*. More specifically, assume y^* denotes the ML segmentation discovered through Viterbi. To build the next-best segmentation, we start from entry $V(T, y^*[T])$. Since that entry is already "consumed" (in the ML segmentation), we trigger a GET-NEXT() call to compute the next-best entry using the entries in row $V(T-1)$; this, in turn, can trigger a GET-NEXT() call at $V(T-1, y^*[T-1])$, and so on. Thus, GET-NEXT() calls are recursively invoked on the path y^* from T down to 1 (or, until we find a $V(i, y^*[i])$ cell with an already-computed "unconsumed" entry).

Complexity: The first GET-NEXT-SEG() call reverts to simple Viterbi ML inference with a complexity of $O(T|Y|^2)$. Then, each incremental call of GET-NEXT-SEG() for the $(L+1)^{th}$ highest-probability segmentation ($L > 0$) has a complexity of $O(T(|Y| + L)\log(|Y| + L))$.

EXAMPLE 4. *The Viterbi V matrix in Figure 2(a) shows the ML segmentation in bold. In order to compute the second highest-probability segmentation,* GET-NEXT*() is triggered recursively from* GET-NEXT-SEG*() to compute the next probable (partial) segmentations entry for cells $V(5,$country$)$, $V(4,$state$)$, $V(3,$city$)$, $V(2,$streetname$)$ and $V(1,$streetname$)$ on the ML path. The second highest-probability segmentation is then computed by backtracking the entry in $V(T, y_T)$ with the second highest score.* □

5.2 Probabilistic Selection

The following query computes the top-1 result of probabilistic selection with condition *streetname contains 'Sacramento'* over the probabilistic *Address* entity table:

```
SELECT *, rank() OVER (ORDER BY prob(*) DESC) r
FROM (  SELECT *
        FROM Address
        WHERE streetname like '%Sacramento%'
) as Address
WHERE r = 1 AND prob(*) > threshold
```

Setting a `threshold` on probability can effectively filter out false positive results with very low probabilities that could be introduced by considering additional non-ML worlds. Please refer to Section 6.3 for a discussion on `threshold` setting.

The probabilistic selection condition is translated into domain constraints that the labels at the positions where $\mathtt{x}_{cond}$ (e.g., "Sacramento") appear in the text-string *d* can only be y_{cond} (e.g., *streetname*). For example, with string "123 Sacramento Street San Francisco CA" and condition *streetname contains 'Sacramento'*, the domain constrains are $D(y_2) = \{streetname\}$ and $D(y_i) = Y$ for $y_1, y_3, .., y_6$. The domain constraints $D(y_1), ..., D(y_n)$ generated from the probabilistic selection condition are used as input to the *constrained Viterbi* algorithm, described in Section 2.3, to compute the top-1 result of probabilistic selection. We illustrate the algorithm using the following example.

EXAMPLE 5. *Consider the V matrix in Figure 2(b) with domain constraints such that y_2 and y_3 can only be streetname: $D(y_2) = D(y_3) = \{streetname\}$. Thus, we can cross out all the cells in $V(2)$ and $V(3)$ where $y \neq$ streetname as shown in Figure 2(b). These domain constraints result in a top-1 constrained path that is different from the result of the vanilla Viterbi algorithm. The top-1 constrained path satisfies the selection condition, and will be returned as a result if its probability is higher than a certain threshold. This constrained top-1 path is the non-ML extraction computed for the probabilistic selection queries.* □

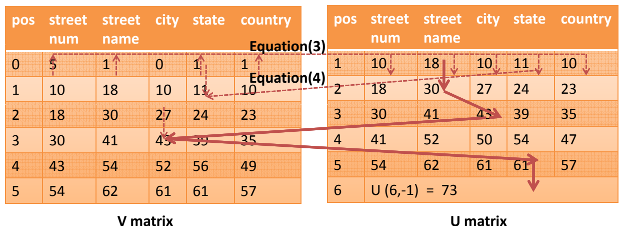
5.3 Probabilistic Join

The following query computes the top-1 (ML) result of a probabilistic join operation over the probabilistic entity tables *Address* and *Company*, with an equality join condition on *city*.

```
SELECT *, rank() OVER (ORDER BY prob(*) DESC) r
FROM (  SELECT *
        FROM Address A, Company C
        WHERE A.city = C.city
) as AddrComp   WHERE r = 1
```

A naive probabilistic join algorithm is: first compute the top-k extractions (for a sufficiently large k) from each input document then deterministically join over the top-k views of the entity tables. The complexity of this algorithm is dominated by the expensive computation of the top-k extractions. Computing a fixed number of top-k extractions for every input document is wasteful, because most top-1 probabilistic join results can be computed from the first few highest-probability extractions. Thus, a more efficient way is to compute the top-1 join results incrementally.

An incremental join algorithm is the rank-join algorithm described in [18]. It takes two ranked inputs and computes the top-k join results incrementally without consuming all inputs. The algorithm maintains a priority queue for buffering all join results that cannot yet be produced, and an upper-bound of the score of all "unseen" join results. Using the *buffer* and the *upper-bound*, the rank-join algorithm can decide if one of the produced join results is guaranteed to be among the top-k answers.



pos	street num	street name	city	state	country
0	5	1	0	1	1
1	10	18	10	11	10
2	18	30	27	24	23
3	30	41	43	39	35
4	43	54	52	56	49
5	54	62	61	61	57

pos	street num	street name	city	state	country
1	10	18	10	11	10
2	18	30	27	24	23
3	30	41	43	39	35
4	41	52	50	54	47
5	54	62	61	61	57
6	U (6,-1) = 73				

Figure 5: Illustration of the data structures in probabilistic projection.

The key idea behind our probabilistic join algorithm is that, given the incremental Viterbi algorithm, which gives us ordered access to the possible extractions of a string *d*—a ranked list by probability—we can compute the top-1 join result for a pair of strings using the rank-join algorithm. Thus, our probabilistic join is a *set of rank-join computations*—one for each string pair that is "joinable" (i.e., can potentially lead to join results). If most "joinable" string pairs compute the top-1 most probable join result from the top-1 extractions, then GET-NEXT-SEG() is called far fewer than k times per document. For the remaining "joinable" string pairs, more extractions are computed incrementally to look for join results in non-ML worlds. In order to bound the search, we set k to be the maximum number of extractions from a document.

If a string pair does not join within the top-k extractions, then rank-join incurs extra overhead. Thus, an effective filter for non-"joinable" string pairs is crucial to the efficiency of the probabilistic join algorithm. We find that an effective such filter is to first compute the top-k extractions for inner (the smaller) entity table, build a hash-table for all the join-key values, and for each string in outer, probe the join-key hash-table for the "joinable" inner strings.

5.4 Probabilistic Projection

The following projection query computes the ML *city* value for each record in the entity table *Address*.

```
SELECT *, rank() OVER (ORDER BY prob(*) DESC) r
FROM (  SELECT city
        FROM Address  ) as Address
WHERE r = 1
```

A naive strategy would be to first compute the full result distribution of a *partial marginalization* over a sub-domain $\chi \subset Y$ for each label y_i in the CRF model (using the distribution of all possible segmentations) [1]; then, the ML world from the distribution can be returned. In this manner, however, significant effort is wasted in calculating the non-ML worlds of the marginal distribution.

Next, we describe a novel probabilistic projection algorithm that integrates the top-1 Viterbi inference and the marginal inference. The key idea underlying our probabilistic projection technique is to extend the Viterbi dynamic program to compute top-1 segmentation paths with labels that are either projected on (i.e., in $Y \backslash \chi$) or *'don't care'*s marginalizing over all the projected-out labels in *chi*.

Similar to vanilla Viterbi, our probabilistic projection algorithm (for projecting out a sub-domain $\chi \subset Y$) also follows a dynamic programming paradigm, computing ML (partial) segmentations ending at position i from those ending at position $i-1$. There are, however, two crucial differences. First, the (partial) segmentations $\{y_1, ..., y_i\}, i \leqslant T$ computed for probabilistic projection only have labels $y_j, 1 \leqslant j \leqslant i$ in $Y \backslash \chi$ or *'don't care'*. y_j =*'don't care'* means that the segmentation path is marginalized over χ at position j (i.e., y_j could be *any* label in χ). A dynamic programming matrix $V(i, y)$ stores such top-k (partial) segmentations that end in

[1] Note that, in contrast to conventional marginal inference (Section 2.3), which marginalizes to eliminate RVs in $\mathbf{y}$, partial marginalization here marginalizes to *restrict the domains* of RVs in $\mathbf{y}$.

$$U(i+1, y_{i+1}).score = \log(\max_{prev}\{\Sigma_{y_i \in \chi}(\bowtie_{prev} (exp(V(i, y_i).score + FactorTbl(x_{i+1}, y_i, y_{i+1}))))\}) \quad (3)$$

$$V(i+1, y_{i+1}).score = \max\{\max_{y_i \in Y \setminus \chi}\{(V(i, y_i).score + FactorTbl(x_{i+1}, y_i, y_{i+1}))\}, U(i+1, y_{i+1}).score\} \quad (4)$$

Figure 4: Equations for computing U and V matrix for probabilistic projection algorithm.

label $y \in Y$ at position i. Second, in order to compute the top-k (partial) segmentations in $V(i+1, y_{i+1})$, in addition to the $V(i, y_i)$ entries, we also need to account for the possibility of a *'don't care'* in position i. This is accomplished by explicitly maintaining an auxiliary matrix $U(i+1, y_{i+1})$ that stores the (partial) top-k segmentations ending at position $i+1$ with label $y_{i+1} \in Y$ and $y_i =$ *'don't care'*. Example V and U matrices for the above example probabilistic projection query are shown in Figure 5.

Next, we describe how the entries in the V and U matrices are computed. Our probabilistic projection algorithm is a variation of the Viterbi technique, where a dynamic programming recurrence computes both V and U simultaneously. Equation (4) in Figure 4 computes the top-1 partial segmentation in cell $V(i+1, y_{i+1})$ by selecting either the best extension of a partial segmentation in $V(i, y_i), y_i \in Y \setminus \chi$ or $U(i+1, y_{i+1})$, depending which choice gives the maximum score. Equation (3) in Figure 4 computes the top-1 partial segmentation for cell $U(i+1, y_{i+1})$, marginalizing over χ at position i, given label $y_{i+1} \in Y$. If we think of each $V(i, y), y \in Y$ as a ranked list of tuples $\{score, label, prev\}$ ordered by descending $score$, then computing the top-ranked $U(i+1, y_{i+1})$ entry is a *multi-way rank-join* with an equality join condition on $prev$ over ranked lists $V(i, y), y \in \chi$. The multi-way rank-join aggregates the entries (i.e., partial segmentations) in multiple lists $V(i, y), y \in \chi$, which agree on $prev$. The join condition (equality on $prev$) ensures that the joined (partial) segmentations share the same path up to position $i-1$. The $score$ of the resulting partial segmentation is the summation over the $score$'s of the joined partial segmentations in $V(i, y), y \in \chi$.

The top-1 result for probabilistic projection can be computed by backtracking the $prev$ pointers from either an entry in $V(T, y_T), y_T \in Y \setminus \chi$ or an aggregated entry $U(T+1, -1)$ computed by aggregating $V(T, y_T), y_T \in \chi$ (to account for a final label of *'don't care'*).

Complexity: Due to the added cost of rank-join aggregations for the U matrix computation, the complexity of our probabilistic projection algorithm is $O(T^3|Y|^3 l)$, where l denotes the depth of the rank-join search.

EXAMPLE 6. *Suppose the query is probabilistic projection over city on Address. Figure 5 illustrates the computation of the entries in the V and U matrices. Equation (3) is used to compute $U(i+1, y'), y' \in Y$ matrix entries from a rank-join over all the $V(i, y)$ lists except when y =city. Equation (4) is used to compute $V(i+1, y)$ matrix entries from the maximum of $V(i, city)$ and $U(i+1, y)$. The top-1 result of probabilistic projection is backtracked from the maximum of $V(5, city)$ and $U(6, -1)$ ($T = 5$) through prev pointers. In this example, the most probable segmentation after probabilistic projection (i.e., partial marginalization) is {'don't care','don't care','don't care',city,'don't care','don't care'}.*

6 Evaluation

Having described two families of approaches for integrating query processing and IE, we now present the results of a set of experiments aimed to evaluate them in terms of efficiency and answer quality.

Setup and Dataset: We implemented the optimizations for select-over-ML (sel-ML) and join-over-ML (join-ML) queries as described in Section 4, and the algorithms for computing the top-1 result for probabilistic selection (prob-sel) and probabilistic join (prob-join) as described in Section 5. These implementations were done on PostgreSQL 8.4.1. We conducted the experiments reported here on a 2.4 GHz Intel Pentium 4 Linux system with 1GB RAM.

For the accuracy experiments, we use the Contact dataset [19] and the CRF model developed at the University of Massachusetts [12]. The Contact dataset contains 182 signature blocks from the Enron email dataset annotated with contact record tags (e.g., *phonenumber*, *city*, *firstname*, etc.). For ground truth, we rely upon manual tagging performed in prior work [12]. We use false positives (i.e., the number of computed results not in the ground truth) and false negatives (i.e., the number of results in ground truth not computed) as the two measures of accuracy.

For the efficiency and scalability experiments we use the DBLP dataset [20] and a CRF model with similar features to those of [12]. The DBLP database contains more than $700k$ papers with attributes, such as *conference*, *year*, etc. We generate bibliography strings from DBLP by concatenating all the attribute values of each paper record. Perhaps because it was generated from structured data, the ML extraction accuracy on the DBLP dataset is as high as 97%, making it not useful for the accuracy experiments. But it serves as an excellent stress test for the scalability of the algorithms.

6.1 Selection over ML world (sel-ML): opt vs. naive

In the first experiment, we use the DBLP dataset to compare the efficiency of optimized and naive algorithms for select-over-ML over $100k$ randomly picked documents (there is no answer quality difference between them — they provide exactly the same answers). We expect the optimized algorithm sel-ML.opt (using SELVITERBI()) to outperform the naive algorithm sel-ML.naive for selection conditions on token sequences that can have different labels in different documents (e.g., 'algorithm' can be in a paper *title* or a *conference* name). Figure 6 shows performance results for sel-ML.naive and sel-ML.opt as the left two bars of each selection condition. It can be seen that the optimized approach can achieve a speedup of 2 to 3 times for 6 selection conditions in Figure 6 [2].

We also ran experiments comparing the two approaches for cases where there is no expected benefit to the sel-ML.opt algorithm (i.e., where there is no pruning position for sel-ML.opt to exploit). As we can see in the three middle selection conditions in Figure 6, the performance of the two approaches was nearly identical, demonstrating that the optimized approach imposes negligible overhead (10% in the worst case) on the selection algorithm.

6.2 Join over ML World (join-ML): opt vs. naive

In the second experiment, we compare the efficiency of the optimized and naive algorithms for join-over-ML. We ran a query over two sets of documents from DBLP to find all pairs in the same *proceeding*. Figure 7 shows the results for different input sizes ($x \times y$ denotes a join of a set of x documents as the inner with a set of y documents as the outer). As can be seen in the figure, join-ML.opt is more than 5 times faster than join-ML.naive when the input sizes are $5k \times 50k$. The speedup of join-ML.opt decreases as the inner size approach the outer size.

Two statistics determine the relative performance of join-ML.opt compared to join-ML.naive: the number of join-keys computed from the inner, and the selectivity of those join-keys over the outer.

[2] It is 10-fold more expensive to evaluate the three right selection conditions, because they appear much more frequently in our dataset.

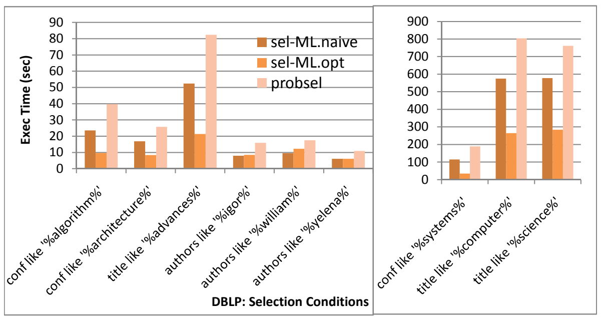


Figure 6: Performance comparison (DBLP) between sel-ML.naive, sel-ML.opt and prob-sel with difference selection conditions.

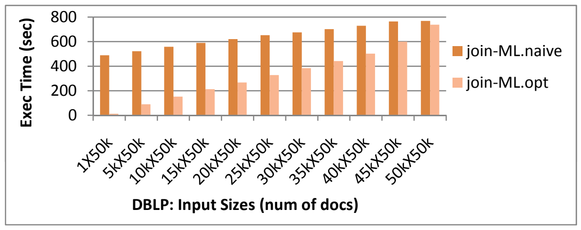


Figure 7: Performance comparison (DBLP) between join-ML.naive and join-ML.opt with different input sizes.

The query joining on "*proceeding*" (shown in Figure 7) is a good case, where it generates a small set of join-keys, which in turn have low selectivity. As the inner size increases, both the number of join-keys and the selectivity increases, hence the speedup decreases.

Other join queries with different values of these two statistics have different performance curves, as shown in Appendix B. There are cases, when the cost of filtering the outer documents using the inner join-keys outweigh the cost of inference over the filtered documents, the join-ML.opt becomes more expensive than join-ML.naive. A query optimizer can estimate the statistics on the size and selectivity of the inner join-keys and determine when it is beneficial to apply join-ML.opt.

In the previous two sections, we showed that the optimizations for SPJ-over-ML queries can achieve significant performance benefits. In the following sections, we turn to probabilistic SPJ queries.

6.3 Probabilistic Selection: prob-sel vs. sel-ML

In this experiment, we compare the answer quality and performance of the probabilistic selection prob-sel and the optimized selection over the ML world sel-ML.opt. We use a set of selection conditions on each of the five labels in the Contact dataset: *companyname*, *firstname*, *lastname*, *jobtitle* and *department*. For each label we use all the values for that label in the ground truth to construct the set of selection conditions, and measure the average false positive and false negative rates. This way, we avoid the randomness due to a particular selection condition.

The table in Figure 8 shows the false negative and false positive rates for the two approaches over the five selection condition sets. As can be seen in the table, the prob-sel algorithm achieves much lower false negatives. For example, the false negatives for

(False -)	company	firstname	lastname	jobtitle	department
sel-ML	0.074	0.012	0.036	0.102	0.286
prob-sel	0.014	0	0.006	0.037	0.079
(False +)	company	firstname	lastname	jobtitle	department
sel-ML	0.010	0	0	0.010	0
prob-sel	0.009	0.006	0.006	0.010	0

Figure 8: False negative, false positive rates (Contact) between sel-ML.opt and prob-sel queries.

(False -)	k=1	k=5	k=10	k=15	k=20
join-ML	0.373	0.373	0.373	0.373	0.373
probjoin.naive	0.373	0.253	0.245	0.216	0.191
probjoin.opt	0.373	0.312	0.283	0.260	0.251
(False +)	k=1	k=5	k=10	k=15	k=20
join-ML	0.082	0.082	0.082	0.082	0.082
probjoin.naive	0.082	0.259	0.437	0.515	0.567
probjoin.opt	0.082	0.091	0.093	0.097	0.110

Figure 9: False negative, false positive rates (Contact) between join-ML.opt and prob-join algorithms.

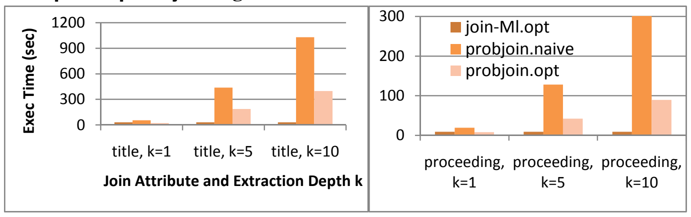


Figure 10: Performance comparison (DBLP) between join-ML.opt and prob-join algorithms.

department is reduced to 30%. The results also show that prob-sel maintains a false positive rate $\leqslant 0.01$ for all the cases.

In many IE application scenarios false negatives are more problematic than false positives because the former means losing answers, and the latter means extra answers, which can be remedied by further processing. Our goal is to reduce the false negative rate without significantly increasing the false positive rate.

Note that in the results shown here, The probability `threshold` in prob-sel algorithm was set to 0.001—the estimated lowest probability of the ML extractions based on a sample documents. Increasing this `threshold` would exclude some ML extractions, which would then increase the false negatives. Decreasing the `threshold` has minimal effect on the false negatives, while increasing the false positive rate slightly. For the queries used in this experiment, all the false positive rates remain below 0.05 even when the `threshold` is set as low as 10^{-6}.

The right most bars for each selection condition in Figure 6 shows the execution time of prob-sel algorithm, compared to sel-ML.naive and sel-ML.opt over $100k$ documents from DBLP. It can be seen that while significantly reducing the false negative rates, the execution time of the prob-sel algorithm is 2 to 5 times that of the corresponding sel-ML.opt algorithm. The is the trade-off between the answer quality and execution time.

We argue that a probabilistic IE system should support both algorithms, so that the right choice can be made based on the requirements of the application.

6.4 Probabilistic Join: prob-join vs. join-ML

Finally, we describe a set of experiments comparing the answer quality and performance of the probabilistic join (prob-join) and the optimized join over the ML world (join-ML.opt). Two prob-join algorithms are prob-join.naive, which joins over top-k worlds, and prob-join.opt, which performs an incremental join as described in Section 5.3.

The experiments for answer quality are conducted on the Contact dataset, and the query is to find all Contact pairs with the same *companyname*. The table in Figure 9 shows that, for both prob-join.opt and prob-join.naive, the false negative rate decreases, and the false positive rate increases as we increase the maximum extraction depth k. prob-join.opt achieves a substantial (slightly over 33%) reduction in false negatives when $k = 20$, with only a moderate increase of the false positive rate. On the other hand, prob-join.naive, although achieving a more significant reduction of the false negative rate, incurs 5 times increase in the false positive rate compared to join-ML.opt. Thus, for most purposes, join-ML.opt achieves the best answer quality among the three algorithms.

The performance experiments are conducted for join queries on *title* and *proceeding* attributes between two document sets with $10k$ and $1k$ documents respectively randomly picked from the DBLP dataset. As shown in Figure 10, for the two join queries with varying settings of k, the join-ML.opt performs significantly better than both prob-join.naive and prob-join.opt when $k > 1$. The optimized algorithm prob-join.opt with the incremental join achieves a significant 2 to 3 times speedup compared to the naive algorithm prob-join.naive. The execution time of both prob-join.naive and prob-join.opt grows linearly with k. Thus, k provides a tuning knob that can be used to adjust the accuracy/performance tradeoff of the prob-join approach.

To summarize, the experiments reported in Section 6.1 and 6.2 demonstrated that integrating the Viterbi inference and relational operators can significantly improve performance of SPJ-over-ML queries without loss of answer quality. The results in Section 6.3 and 6.4 showed that the answer quality (i.e. false negatives) can be improved significantly by probabilistic SPJ queries. The answer quality comes in some cases (e.g., probabilistic selection) at a moderate cost, and in others (e.g., probabilistic join) at a higher cost. One advantage of representing the IE model and inference algorithms in database is that a cost-based query optimizer can choose between the different execution techniques explored by this paper.

7 Related Work

Information extraction (IE) has received a lot of attention from both the database and the Machine Learning communities (see recent tutorials [21, 22]). In the database community, significant effort has been devoted to exploring *frameworks* that manage the state of the IE process, and simplify the specification, optimization and execution of IE tasks. A promising approach in this line of work is to use *declarative* specifications of the IE tasks [1, 2, 3]. These earlier efforts in declarative IE, however, lack a unified framework supporting both a declarative interface as well as the state-of-the-art probabilistic IE models. There have been efforts to consider uncertainties in IE, but only in limited settings [23, 24].

The vast majority of IE work in Machine Learning focuses on improving extraction accuracy using probabilistic techniques, including variants of HMM [25] and CRF [12, 14, 15] models. At the same time, the Machine Learning community has also been moving in the direction of declarative IE [26, 27] using first-order logic.

Since the early 80's, a number of PDBSs have been proposed in an effort to offer a declarative, SQL-like interface for managing large uncertain-data repositories [4, 5, 6, 7, 8, 9]. Recent PDBS efforts such as BAYESSTORE [9] and the work of Sen and Deshpande [7] represent probabilistic models (based on Bayesian networks) as first-class citizens in a relational database, and support in-database queries and reasoning over the model.

The issue of providing database support for managing IE based on probabilistic models has not been addressed in existing PDBSs. Closer to our work, Gupta and Sarawagi [11] give tools for storing coarse approximations of a CRF model inside a simple PDBS supporting limited forms of dependencies. Instead, our work aims to support the full expressive power of CRF models and inference operations as a first-class PDBS citizen.

8 Conclusions

The need for query processing over data extracted from unstructured sources is becoming increasingly acute. Previous approaches to integrating IE with database systems suffered from both performance and accuracy limitations. In this paper we proposed two families of queries over CRF-based IE results. The first uses deterministic queries over maximum-likelihood extractions, with optimizations to push the relational operators into the Viterbi algorithm. The second extends the Viterbi algorithm to produce a set of possible extraction "worlds", from which we compute top-k probabilistic query answers. These approaches provide a design space in which answer quality and performance can be traded off according to the needs of the application. As future work, we intend to explore additional performance enhancements via parallelization of the algorithms and the development of a cost-based optimizer.

Acknowledgements: This work was supported by the NSF, under grant NSF IIS-0415175 and IIS-0803690, and a Google Faculty Research Award. We wish to thank Prof. Aron Culotta of SELU, who provided assistance with the Contact dataset and the corresponding CRF model [19].

9 References

[1] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan, "An Algebraic Approach to Rule-Based Information Extraction," in *ICDE*, 2008.

[2] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan, "Declarative Information Extraction Using Datalog with Embedded Extraction Predicates," in VLDB, 2007.

[3] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen, "Community information management," 2006.

[4] N. Dalvi and D. Suciu, "Efficient Query Evaluation on Probabilistic Databases," in *VLDB*, 2004.

[5] O. Benjelloun, A. Sarma, A. Halevy, and J. Widom, "ULDB: Databases with Uncertainty and Lineage," in *VLDB*, 2006.

[6] A. Deshpande and S. Madden, "MauveDB: Supporting Model-based User Views in Database Systems," in *SIGMOD*, 2006.

[7] P. Sen and A. Deshpande, "Representing and Querying Correlated Tuples in Probabilistic Databases," in *ICDE*, 2007.

[8] L. Antova, T. Jansen, C. Koch, and D. Olteanu, "Fast and Simple Relational Processing of Uncertain Data," in *ICDE*, 2008.

[9] D. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein, "BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models," in *VLDB*, 2008.

[10] D. Wang, E. Michelakis, M. Franklin, M. Garofalakis, and J. Hellerstein, "Probabilistic Declarative Information Extraction," in *ICDE*, 2010.

[11] R. Gupta and S. Sarawagi, "Creating Probabilistic Databases from Information Extraction Models," in *VLDB*, 2006.

[12] T. Kristjansson, A. Culotta, P. Viola, and A. McCallum, "Interactive Information Extraction with Constrained Conditional Random Fields," in *AAAI'04*, 2004.

[13] "Enron email dataset, http://www.cs.cmu.edu/ enron/."

[14] J. Lafferty, A. McCallum, and F. Pereira, "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data," in *ICML*, 2001.

[15] C. Sutton and A. McCallum, "Introduction to Conditional Random Fields for Relational Learning," in *Introduction to Statistical Relational Learning*, 2008.

[16] G. D. Forney, "The Viterbi Algorithm," *IEEE*, 1973.

[17] L. Huang and D. Chiang, "Better k-best Parsing," in *IWPT*, 2005.

[18] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid, "Rank-aware Query Optimization," in *SIGMOD*, 2004.

[19] "Contact record extraction data, http://www2.selu.edu/academics/faculty/aculotta/data/contact.html."

[20] "Dblp dataset, http://kdl.cs.umass.edu/data/dblp/dblp-info.html."

[21] E. Agichtein and S. Sarawagi, "Scalable Information Extraction and Integration," in *KDD*, 2006.

[22] A. Doan, R. Ramakrishnan, and S. Vaithyanathan., "Managing Information Extraction: State of the Art and Research Directions," in *SIGMOD*, 2006.

[23] E. Michelakis, P. Haas, R. Krishnamurthy, and S. Vaithyanathan, "Uncertainty Management in Rule-based Information Extraction Systems," in *SIGMOD*, 2009.

[24] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan, "Toward best-effort information extraction," in SIGMOD, 2008.

[25] M. Skounakis, M. Craven, and S. Ray, "Hierarchical Hidden Markov Models for Information Extraction," in *Proc. of IJCAI*, 2003.

[26] J. Eisner, E. Goldlust, and N. Smith, "Compiling Comp Ling: Practical Weighted Dynamic Programming and the Dyna," in *HLT/EMNLP*, 2005.

[27] H. Poon and P. Domingos, "Joint Inference in Information Extraction," in *Proc. of AAAI*, 2007.

APPENDIX

A Pseudo-codes and Complexity Analysis

In this section, we include the detailed descriptions of the pseudo-code and the complexity analysis for some of the key query processing algorithms discussed in the paper, including SELVITERBI (Figure 12), incremental Viterbi algorithm (Figure 13), probabilistic join algorithm (Figure 16), and probabilistic projection algorithm (Figures 14, 15).

A.1 SELVITERBI

The SELVITERBI algorithm in Figure 12 follows the vanilla Viterbi dynamic programming algorithm, which calls a User Defined Function (UDF), at each recursive step i, to compute the top-1 (partial) segmentations in $V(i, y_i)$ using Equation (2), with additional logic to check if i is the smallest pruning position. It stops the recurrence and returns *null* if a pruning position is found.

Generalized Selection-Aware Viterbi: The SELVITERBI algorithm can be used with any $\{i, len, y'\}$ condition. However, for conditions with i being the last few positions, a variant of the Viterbi algorithm, which computes the top-k segmentations from the end to the start of a text-string, should be used for efficiency. Furthermore, the Viterbi algorithm can be modified to start the dynamic programming from any position i, by expanding the intermediate result of the partial segmentations in two directions, rather than just one (left to right) in Viterbi. As future work, a cost-based optimizer can be developed to decide where/how to execute SELVITERBI, based on the position and selectivity of a selection condition.

A.2 Incremental Viterbi Algorithm

The incremental Viterbi algorithm in Figure 13 takes the document doc, the current V matrix, the last top segmentation y^*, and the number of top segmentations generated so far L, to compute the next highest-probability segmentation y^*_{new}. If $L = 0$, then GET-NEXT-SEG() calls VITERBI() to compute the ML segmentation. If $L > 0$, then recursive GET-NEXT() calls are triggered for V cells on the y^* path. The chain of GET-NEXT() calls stops either at row 1 or when $y^*[i]$ is not the last entry in cell $V(i, y^*[i])$. Equation (2) is used to compute the next probable (partial) segmentation in $V(i, y^*[i])$ after computing the new entry for $V(i-1, y^*[i-1])$. Finally, GET-NEXT-SEG() backtracks the next highest-probability segmentation y^*_{new} from the $(L+1)$th maximal entry in row $V(T)$.

Complexity: When $L = 0$, GET-NEXT-SEG() calls VITERBI(), thus the complexity is $O(T|Y|^2)$. When $L > 0$, the complexity of the incremental Viterbi algorithm GET-NEXT-SEG() is $O(T(|Y| + L)log(|Y|+L))$, because GET-NEXT() is called maximum T times, where T is the length of the document. Each GET-NEXT() call needs to sort at most $(|Y|+L)$ entries, because each cell in $V(i, y_i)$ contains 1 entry after the VITERBI() ML inference, and at most 1 entry is added to 1 cell in $V(i, y_i)$ for each GET-NEXT-SEG() call.

A.3 Probabilistic Join Algorithm

The probabilistic equijoin algorithm in Figure 16 takes two joining relations: $inner$, $outer$, two joining columns: $incol$ and $outcol$, and a parameter k indicating an upper bound on the number of extractions allowed per document. The algorithm first computes the top-k segmentations for each $inner$ document, which overlaps with at least one document in $outer$, using the incremental Viterbi algorithm.

For each string $odoc \in outer$, which contains at least one join-key from an $inner$ extraction, we compute the corresponding $joinable$ set—a set of inner documents that have a join-key contained in $odoc$. Then, the rank-join loop starts simultaneously for all $idoc \in joinable$ and $odoc$ pairs. The next highest-probability segmentation $yout^*$ for $odoc$ is computed. All $inner$ segmentations that have join-key value $yout^*.outcol$ are retrieved, and the joined results are inserted into $matched$—a hash-table containing all the matching $\{idoc, odoc\}$ pairs and their corresponding rank-join buffer containing matching segmentations $\{yin^*, yout^*\}$. Next, the upper-bound $upper$ for each $\{idoc, odoc\}$ pair in $matched$ is computed, and the segmentation pairs $\{yin^*, yout^*\}$ that have probability no less than $upper$ are returned. Finally, $idoc$ and $\{idoc, odoc\}$ are deleted from $joinable$ and $matched$, respectively. The rank-join loop stops when either the $joinable$ set becomes empty or the extraction depth exceeds k.

A.4 Probabilistic Projection Algorithm

The detailed pseudo-code descriptions of probabilistic projection algorithm can be found in Figure 14.

Complexity: The complexity of the probabilistic projection algorithm is $O(T^3|Y|^3l)$, where l is the average depth of the rank-join algorithm in GET-NEXT-U(). The additional complexity comes from computing the U matrix. The computation of each entry in the $U(i, y_{i+1})$ matrix may recursively triggers up to T number of GET-NEXT-U() calls. And each GET-NEXT-U() is a rank-join algorithm, which calls MARGINALIZATION() with complexity $O(T|Y|^2)$ in each iteration. Thus, we have the complexity being $O(T^3|Y|^3l)$.

B Additional Evaluation

Join-ML.naive vs. Join-ML.opt: This is the additional results for the experiment comparing the efficiency of the optimized and naive algorithms for join-over-ML. We ran a query over two sets of documents from the DBLP to find all pairs with the same *authors*. Figure 11(a) shows the results for different input sizes. As can be seen in the figure, join-ML.opt is more efficient than join-ML.naive when the inner size is much lower than outer size, and join-ML.opt grows to be more expensive as the inner size increases. For this join query, the size of inner join-keys is large and the selectivity of those join-keys is high.

Similarly, we ran the query joining on *publisher*. Since there are not many publishers, the size of the inner join-keys remain small as the size of the inner increases, while the selectivity of those join-keys is high. Thus, as can be seen in Figure 11(b), join-ML.opt have similar or slightly better performance compare to join-ML.naive for all inner sizes.

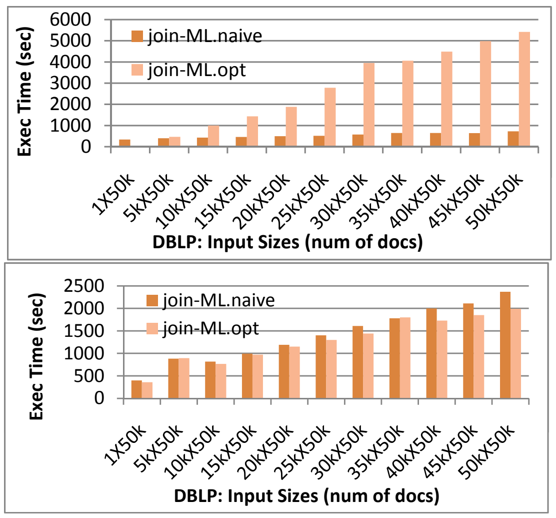


Figure 11: **Performance comparison (DBLP) between join-ML.naive and join-ML.opt with different input sizes.**

```
CREATE FUNCTION selViterbi (id, i, len, y') RETURN VOID AS
$$
-- compute the V matrix from tokenTbl and factorTbl
INSERT INTO V
WITH RECURSIVE Vtop1(pos, entries, filter) AS (
     SELECT st.pos,
        mr.score[1:|Y|]||array-fill(-1,ARRAY[|Y|]) entries,
        array-fill(0,ARRAY[|Y|]) filter
     FROM tokenTbl st, factorTbl mr
     WHERE st.strID=id AND st.pos=0 AND mr.segID=st.segID
     UNION ALL
     SELECT st.pos,
        top1-array(v.score, mr.score) entries,
        update-filter(i,len,y',v.pos,v.filter,v.score) filter
     FROM tokenTbl st, Vtop1 v, factorTbl mr
     WHERE st.strID=id AND st.pos = v.pos+1 AND
        mr.segID=st.segID AND (pos<=i or nonzero(v.filter))
) SELECT pos, entries FROM Vtop1,
-- backtrack the ML segmentation from V
INSERT INTO Ptop1
WITH RECURSIVE P(pos,entry) AS (
      SELECT pos, get-max(entries) entry
      FROM Vtop1 WHERE pos=T-1
      UNION ALL
      SELECT V.pos, V.entries[P.prevLabel]
      FROM Vtop1 V, P WHERE V.pos = P.pos-1
) SELECT id as strID, pos, entry.label FROM Ptop1
$$
LANGUAGE SQL;
```

```
UPDATE-FILTER $(i, len, y', j, filter, V(j,y))$
1  for each $l \in Y$ do
2     if $j \geqslant (i + len) \&\& filter[V(j,l).prev] = 1$ then
3        $newFilter[l] = 1$;
4     else if $j > i \&\& l = y' \&\& filter[V(i-1,l).prev] = 1$ then
5        $newFilter[l] = 1$;
6     else if $j = i \&\& l = y'$ then
7        $newFilter[l] = 1$;
8     else $newFilter[l] = 0$;
9  endif endfor
10 return $newFilter$
```

Figure 12: **SELVITERBI(): optimized algorithm for select-over-ML queries, which stores the ML segmentation in Ptop1. UPDATE-FILTER(): auxiliary function for checking pruning position.**

```
GET-NEXT-SEG $(doc, V, y^*, L)$
1  if $L = 0$ then
2     $y^*_{new} \leftarrow$ VITERBI$(doc)$;
3  else
4     $T \leftarrow doc.length$;
5     // compute the next best entry for cell $V(T, y^*[T])$
6     GET-NEXT$(V, T, y^*[T])$;
7     $Tmp \leftarrow \cup_{y_T \in Y} V(T, y_T)$;
8     sort $Tmp$ by score desc;
9     $y^*_{new}[n] \leftarrow Tmp[L+1]$;
10    for $i = n-1; i > 0; i--$ do
11       $y^*_{new}[i] \leftarrow y^*_{new}[i+1].prev$;
12 endfor endif
13 return $y^*_{new}$
GET-NEXT $(V, i, e)$
1  $maxidx \leftarrow V(i, e.label).length; l \leftarrow e.label$;
2  if $e = V(i,l)[maxidx]$ then
3     GET-NEXT$(V, i-1, y^*[i-1])$;
4     $Tmp \leftarrow \cup_{y_{i-1} \in Y} V(i-1, y_{i-1})$;
5     for each $te = \{score, label, prev\} \in Tmp$ do
6        $te.score \leftarrow te.score +$ FACTORTBL$(l, te.label, x_i)$;
7     endfor
8     sort $Tmp$ by score desc;
9     $V(i,l) \leftarrow V(i,l) \cup Tmp[maxidx+1]$;
10 endif
```

Figure 13: **The incremental Viterbi algorithm for getting the next highest-probability segmentation.**

```
TOP1PROBPROJECT $(doc, \chi)$
1  $V(1, y_1)[1].score \leftarrow$ FACTORTBL$(doc.x_1, y_1, -1)$;
2  $V(1, y_1)[1].prev = \{-1, -1\}$;
3  for each token $doc.x_i (i = 2, ..., T)$ do
4     for each cur label $y \in Y$ do
5        GET-NEXT-U$(V, i, y, \chi)$;
6        GET-NEXT-V$(V, i, y, \chi)$;
7  endfor endfor
8  $Tmp \leftarrow \cup_{y_T \in Y} V(T, y_T)$;
9  $Tmp \leftarrow Tmp \cup$ GET-NEXT-U$(V, T+1, -1, \chi)$;
10 sort $Tmp$ by score desc; $y^*[T] \leftarrow Tmp[1]$;
11 for $i = T-1; i >= 0; i--$ do
12    $label \leftarrow y^*[i+1].prev.label$;
13    if $label =$ 'don't care' then
14       $y^*[i] \leftarrow U(i, y^*[i+1].label)[1]$;
15    else $y^*[i] \leftarrow V(i, label)[1]$;
16 endif endfor
17 return $y^*$
GET-NEXT-V $(V, i, y, \chi)$
1  $maxidx \leftarrow V(i,y).length$;
2  if $maxidx > 0$ then
3     $py \leftarrow V(i,y).prev.label; pidx \leftarrow V(i,y).prev.idx$;
4     if $py =$ 'don't care' $\&\& pidx = U(i,y).length$ then
5        GET-NEXT-U$(V, i, y, \chi)$;
6     else if $py \neq$ 'don't care' $\&\& pidx = V(i-1, py).length$ then
7        GET-NEXT-V$(V, i-1, py, \chi)$;
8  endif endif
9  $Tmp \leftarrow \cup_{y' \in Y \setminus \chi} V(i-1, y')$;
10 for each $te = \{score, label, prev\} \in Tmp$ do
11    $te.score \leftarrow te.score +$ FACTORTBL$(x_i, te.label, y')$;
12 endfor
13 $Tmp \leftarrow Tmp \cup U(i,y)$;
14 sort $Tmp$ by score desc;
15 $V(i,y) \leftarrow V(i,y) \cup Tmp[maxidx+1]$;
```

Figure 14: **Algorithm for computing top-1 result for probabilistic projection queries.**

```
GET-NEXT-U $(V, i, y, \chi)$
    $maxidx \leftarrow U(i, y).length; labels[i] \leftarrow \chi;$
    for $l = 1; true; l++$ do
        $upper \leftarrow \Sigma_{y \in \chi} V(i, y').\text{GET-LAST}().score;$
        // $UT$ is the rank-join buffer
        $score \leftarrow UT(i, y)[1].score;$
        if $upper \leqslant score$ then
            $U(i, y)[maxidx + 1] \leftarrow \{score, y, UT(i, y)[1].prev\};$
            $UT(i, y) \leftarrow UT(i, y) - UT(i, y)[1]; break;$
        endif
        $Tmp \leftarrow \cup_{y' \in \chi} V(i-1, y').\text{GET-LAST}();$
        for each $te = \{score, label, prev\} \in Tmp$ do
            $te.score \leftarrow te.score + \text{FACTORTBL}(x_i, te.label, y');$
        endfor
        sort $Tmp$ by score desc; $max \leftarrow Tmp[1];$
        for $j = i - 2; j >= 0; j--$ do
            if $max.prev.label =$ 'don't care' then
                $max \leftarrow U(j, max.label)[max.prev.idx];$
                $label[j] \leftarrow \chi;$
            else
                $max \leftarrow V(j, max.prev.label)[max.prev.idx];$
                $label[j] \leftarrow max.prev.label;$
        endif endfor
        $score \leftarrow \text{MARGINALIZE}(i, labels);$
        $UT(i, y) \leftarrow UT(i, y) \cup \{score, y, Tmp[1].prev\};$
        $\text{GET-NEXT-V}(V, i-1, Tmp[1].label, \chi);$
    endfor
```

Figure 15: **Algorithm for computing the next highest-probability entry in $U(i, y)$ used in TOP1PROBPROJECT().**

```
TOP1PROBJOIN $(inner, outer, k, incol, outcol)$
    for each $idoc \in inner$ that overlap with at least one $odoc \in outer$ do
        $V \leftarrow null; yin^* \leftarrow null;$
        for $i = 0; i < k; i{+}{+}$ do
            $yin^* \leftarrow \text{GET-NEXT-SEG}(idoc, V, yin^*, i);$
            $inTopk.\text{PUT}(\{idoc, i\}, yin^*);$
            $inKeys.\text{PUT}(yin^*.incol, \{idoc, i\});$
    endfor endfor
    for each $odoc \in outer$ that contains at least one $inKeys.keys$ do
        $V \leftarrow null; yout^* \leftarrow null;$
        $joinable \leftarrow$ all idoc that has $inKeys.keys$ contained in odoc
        for $i = 0; i < k; i++$ do
            $yout^* \leftarrow \text{GET-NEXT-SEG}(odoc, V, yout^*, i)$
            $outTopk.\text{PUT}(\{odoc, i\}, yout^*);$
            for each $\{idoc, idx\} \in inKeys.\text{GET}(yout^*.outcol)$ do
                if $idoc \in joinable$ then
                    $yin^* \leftarrow inTopk.\text{LOOKUP}(\{idoc, idx\});$
                    $matched.\text{PUT}(\{idoc, odoc\}, \{yin^*, yout^*\})$
            endif endfor
            for each doc pair $\{idoc, odoc\} \in matched$ do
                $upper \leftarrow outTopk(odoc).maxprob \times inTopk(idoc).minprob;$
                $upper \leftarrow \max(yout^*.prob \times inTopk(idoc).maxprob, upper);$
                $prob \leftarrow yin^*.prob \times yout^*.prob$
                if $prob \geqslant upper$ then
                    return next $\{idoc, odoc, yin^*, yout^*\}$
                    $matched.\text{DELETE}(\{idoc, odoc\});$
                    $joinable.\text{DELETE}(idoc);$
            endif endfor
            if $joinable = \emptyset$ then $break;$
    endif endfor endfor
```

Figure 16: **Algorithm for computing the top-1 result for probabilistic join queries.**