Declarative Information Extraction in a Probabilistic Database System

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

ABSTRACT

Full-text documents represent a large fraction of the world's data. Although not structured per se, they often contain snippets of structured information within them: e.g., names, addresses, and document titles. Information Extraction (IE) techniques identify such structured information in text. In recent years, database research has pursued IE on two fronts: declarative languages and systems for managing IE tasks, and IE as an uncertain data source for Probabilistic Databases. It is natural to consider merging these two directions, but efforts to do so have had to compromise on the statistical robustness of IE algorithms in order to fit with early Probabilistic Database models.

In this paper, we bridge the gap between these ideas by implementing a state-of-the-art statistical IE approach – Conditional Random Fields (CRFs) – in the setting of Probabilistic Databases that treat statistical models as first-class data objects. Using standard relational tables to capture CRF parameters, and inverted-file representations of text, we show that the Viterbi algorithm for CRF inference can be specified declaratively in recursive SQL, in a manner that can both choose likely segmentations, and provide detailed marginal distributions for label assignment. Given this implementation, we propose query processing optimizations that effectively combine probabilistic inference and relational operators such as selections and joins. In an experimental study with two data sets, we demonstrate the efficiency of our in-database Viterbi implementation in PostgreSQL relative to an open-source CRF library, and show the performance benefits of our optimizations.

1 Introduction

The field of database management has traditionally focused on *structured* data, leaving unstructured textual data largely in the hands of other research communities including Information Retrieval, AI, and Web technologies. Recently, however, there has been significant interest across all of these areas in techniques that parse text and extract structured objects that can be integrated into traditional databases. This task is known as Information Extraction (IE).

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 for the task of IE [21, 24]. Viewing the steps of IE algorithms through the lens of query languages and query processing, it is possible to cleanly abstract various IE tasks and their compositions, and improve their performance via query optimization techniques. Second, IE has been a major motivating application for the recent groundswell of work on Probabilistic Databases (PDBS) [7, 4, 8, 23, 2, 28], which can model the uncertainty inherent in IE outputs, and enable users to write declarative queries that reason about that uncertainty.

Given this background, it is natural to consider merging these two ideas into a single architecture: a unified database system that enables well-specified IE tasks, and provides a probabilistic framework for querying the outputs of those tasks. This is especially natural for leading IE approaches like Conditional Random Fields (CRFs) [17] that are themselves probabilistic machine learning methods. The query language of the PDBS should be able to capture the models and methods inherent in these probabilistic IE techniques.

Gupta and Sarawagi recently considered exactly this issue: implementing CRFs in the context of a probabilistic database [14]. However, they used a PDBS model that only supported very limited forms of *tuple-* and *attribute-level uncertainty*, and made strong independence assumptions across tuples/values. These limitations enabled them to capture only a coarse approximation of the CRF distribution model inside the PDBS.

Inspired by that work, in this paper we show how to bridge the gap between CRF-based IE and probabilistic databases, preserving the fidelity of the CRF distribution while enabling opportunities for query optimization. Our technique is based on the following observations:

1. *Relational Representation of Inputs:* CRFs can be very naturally modeled as first-class data in a relational database, in the spirit of recent PDBS like BayesStore [28] and the work of Sen and Deshpande [23]. Similarly, text data can be captured relationally via the inverted file representation commonly used in information retrieval.
2. *Declarative Viterbi Inference:* Given tabular representations of CRF model parameters and input text, the central algorithm for CRFs – Viterbi inference [11] – can be elegantly expressed as a standard recursive SQL query for dynamic programming.
3. *Query Optimization:* The relational representations of the probabilistic model and inference algorithm lead to query optimization opportunities, and integrate naturally to populate the uncertain attributes of a PDBS.

Together, these result in a unified and efficient approach for implementing CRF in a database engine, providing a flexible, principled foundation for subsequent probabilistic querying. Importantly, our approach not only correctly performs CRF-based IE on input text, it

Submitted for publication. Version as of 3/20/2009. Contact author: D. Z. Wang.

also maintains the probability distributions inherent in CRF to enable a standard *possible worlds* semantics for the PDBS. We have implemented these ideas in the PostgreSQL DBMS, and show performance benefits relative to a standalone open-source Viterbi implementation.

2 Background

This section covers the concept of a probabilistic database, the declarative information extraction, the CRF model, 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 $pwd(D^p)$. 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 $\mathcal{R}$.

There has been significant work recently [7, 4, 8, 23, 2, 28], proposing different ways to represent the uncertainties in data and the probability distribution over possible worlds. To represent the uncertainties in data, some approaches associate the uncertainties to tuples, while others associate them to values. To represent the probability distribution over $pwd(D^p)$, some systems use boolean expressions of probabilistic events, others support statistical or machine learning models.

2.2 Declarative Information Extraction

Information extraction (IE) has been one of the key driver applications for probabilistic databases. IE is the task of automatically extracting and labeling structured entities in unstructured data sources, such as newswire documents, emails and the Web. The databases generated from an IE process contain uncertain data, because despite the success of the techniques developed for IE, the accuracy of the extracted data is inherently imperfect.

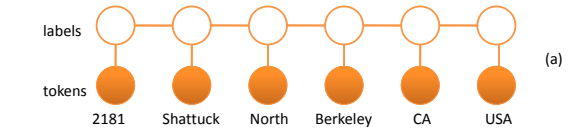
More recently, *declarative information extraction* [21, 24] has been proposed to build database systems providing a declarative programming interface and cost-based query optimization for building IE applications.

In addition to extracting and labeling entities in text, it is highly desirable for an IE system to provide a probability distribution over the extracted labels to 1) enable exploration of accuracy-coverage trade-offs to improve data integrity in database; 2) provide confidence of the extracted data for interactive information extraction; 3) improve the performance of data mining algorithms that use databases created by IE systems.

2.3 Conditional Random Fields (CRF)

The declarative IE system we build and evaluate is based on a linear-chain Conditional Random Field (CRF) [17, 26], which is a state-of-the-art probabilistic model for solving IE tasks. The CRF model, which is closely related to Hidden Markov Models (HMMs), has performed well on IE tasks because of its ability to capture arbitrary, overlapping features of the input in a Markov model.

A CRF model represents the probability distribution over sets of random variables $\mathbf{V} = \mathbf{X} \cup \mathbf{Y}$, where $\mathbf{X}$ is a set of input variables that we assume are observed and $\mathbf{Y}$ is a set of output variables that we wish to predict. In the context of information extraction, $\mathbf{X}$ is a sequence of tokens in a document, and $\mathbf{Y}$ is a sequence of corresponding labels. We denote an assignment to $\mathbf{X}$ by $\mathbf{x}$ and to $\mathbf{Y}$ by $\mathbf{y}$.



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

(b)

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

(c)

	apt. num	street num	street name	city	state	country	prob
y1	null	2181	Shattuck North	Berkeley	CA	USA	0.6
y2	2181	null	Shattuck	North Berkeley	CA	USA	0.1

(d)

Figure 1: (a) Example CRF model for a address string; (b) A sample of the TOKENTBL table; (c) a sample of the MR table; (d) two possible segmentations $\mathbf{y}_1$, $\mathbf{y}_2$.

We now describe a running example which we will use throughout this paper. The example IE task is called *field segmentation*, in which a document is regarded as a sequence of pertinent fields, and the goal is to tag each token in a document with one of the field labels. For example, field segmentation can be applied to extract *street address, city, country* information from *address* strings, and *author, paper, journal* information from *bibliographic* citations.

EXAMPLE 1. *Figure 1(a) shows a CRF model over an address string $\mathbf{x}$'2181 Shattuck North Berkeley CA USA'. The possible labels are Y = {apt. num, street num, street name, city, state, country}. A segmentation $\mathbf{y} = \{y_1, ..., y_T\}$ is one possible way to tag each tokens in $\mathbf{x}$ into one of the field labels in Y.*

The following definition defines the conditional probabilistic distribution of $\mathbf{y}$ given a specific assignment $\mathbf{x}$ by the CRF model.

DEFINITION 2.1. *Let $\mathbf{X}$, $\mathbf{Y}$ be random vectors, $\Lambda = \{\lambda_k\} \in R^K$ be a parameter vector, and $\{f_k(y_t, y_{t-1}, x_t)\}_{k=1}^K$ be a set of real-valued feature functions. Then a linear-chain conditional random field is a distribution $p(\mathbf{y} \mid \mathbf{x})$ that takes the form:*

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{1}{Z(\mathbf{x})} exp\{\sum_{t=1}^{T}\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t)\}, \quad (1)$$

where $Z(\mathbf{x})$ is an instance-specific normalization function

$$Z(\mathbf{x}) = \sum_{\mathbf{y}} exp\{\sum_{t=1}^{T}\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t)\}. \quad (2)$$

Correlations in the CRF model exist only between neighboring labels (e.g., between y_t and y_{t-1} and between the token x_t and its label y_t). Such correlations are represented by the feature functions $\{f_k(y_t, y_{t-1}, x_t)\}_{k=1}^K$. In the context of Example 1, here are two possible feature functions:

$$\begin{aligned} f_1(y_t, y_{t-1}, x_t) &= [x_t \textit{ appears in a city list}] \cdot [y_t = \textit{city}] \\ f_2(y_t, y_{t-1}, x_t) &= [x_t \textit{ is an integer}] \cdot [y_t = \textit{apt. num}] \\ &\quad \cdot [y_{t-1} = \textit{street name}] \end{aligned}$$

A CRF model represents the probability distribution over all possible segmentations $\mathbf{Y}$ of a document d. Figure 1(d) shows two

possible segmentations of *d* and their probabilities, where the probabilities of all possible segmentations should sum up to 1: $\sum_{\mathbf{Y}} p(\mathbf{y}|\mathbf{x}) = 1$. These possible segmentations are the "possible worlds" for a particular document *d*.

2.4 Inference Queries on CRF Model

There are three main types of inference queries over the CRF model [26]: (1) top-k inference; (2) constrained top-k inference; and (3) marginal inference.

2.4.1 Top-k Inference

Top-k inference is the most frequently used type of inference query over the CRF model. It determines the *segmentations* with the top-k highest probabilities given a token sequence $\mathbf{x}$ from a document *d*.

The *Viterbi* dynamic programming algorithm [11] is a key implementation technique for top-k inference in IE applications. For simplicity, we provide the equations to compute the top-1 segmentation. These can be easily extended to compute the top-k. The dynamic programming algorithm computes a two dimensional V matrix, where each cell $V(i, y)$ stores the top-1 partial segmentations ending at position i with label y.

$$V(i,y) = \begin{cases} \max_{y'}(V(i-1,y') \\ \quad + \sum_{k=1}^{K} \lambda_k f_k(y, y', x_i)), & if \quad i \geqslant 0 \\ 0, & if \quad i = -1. \end{cases} \tag{3}$$

We can backtrack through the V matrix to recover the top-1 segmentation sequence $y*$. The complexity of the Viterbi algorithm is $O(T \cdot |Y|^2)$ where T is the length of the document and $|Y|$ is the number of labels.

2.4.2 Constrained Top-k Inference

Constrained top-k inference [16] is a special case of top-k inference. It is used when a subset of the token labels has been provided, for example via a user interface, or application specific meta-data (e.g., an email header that deterministically labels tokens). Let $\mathbf{s}$ be the evidence vector $\mathbf{s} = \{s_1, ..., s_T\}$, where s_i stores the label evidence for position i. $s_i = \emptyset$ when there is no evidence for position i, and $s_i \in Y$ when evidence exists. The constrained top-k inference can be computed by the Constrained Viterbi algorithm using the following equation:

$$V(i,y) = \begin{cases} \max_{y'}(V(i-1,y') \\ \quad + \sum_{t=1}^{T} \sum_{k=1}^{K} \lambda_k f_k(y, y', x_i)), & if \quad s_i = \emptyset \mid y = s_i \\ 0, & if \quad s_i \neq \emptyset \& y \neq s_i \end{cases} \tag{4}$$

2.4.3 Marginal Inference

A third inference task that is useful in some applications is to compute a marginal probability $p(y_t, y_{t+1}, ..., y_{t+k}|\mathbf{x})$ over a single label or a sub-sequence of labels in a document *d*. This type of inference is useful for estimating the confidence of one or a set of extracted fields, so that, for example, the low-confidence extractions can be highlighted to solicit user feedback [16], to enable users to explore accuracy-coverage trade-off based on confidence, or to enable data mining tools to query over the probability distributions over data [6].

Queries on the marginals form the strongest motivation for having a probabilistic database maintain the full distribution over the extracted data, rather than just the top-k inference results. In Section 4.3, we briefly discuss how to extend the SQL implementation for the Viterbi algorithm to compute the marginal inference queries.

3 CRF Models in a Probabilistic Databases

Although IE has been cited as a key driving application for probabilistic database research, to date there has been a gap between probabilistic IE and PDBSs.

In this section, we describe the design of a probabilistic database $\mathcal{DB}^p$=<$\mathcal{R}$, F> that can support rich probabilistic IE models, such as CRF by (1) storing documents in an incomplete relation R as an inverted file with a probabilistic label^p attribute; and (2) storing the exact probability distribution F over the extraction possibilities from CRF through a factor table.

3.1 Token Table: The Incomplete Relation

The token table TOKENTBL is an incomplete relation R in $\mathcal{DB}^p$, which stores documents as relations in a database, in a manner akin to the inverted files commonly used in information retrieval. As shown in Figure 1(b), each tuple in TOKENTBL records a unique occurrence of a token, which is identified by the document ID (docID) and the position (pos) the token is taken from. A TOKENTBL has the following schema:

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

The TOKENTBL contains one probabilistic attribute – label^p, which can contain missing values, whose probability distribution will be computed from the CRF model. The deterministic attributes of the TOKENTBL are populated by parsing the input documents $\mathcal{D}$, with label values marked as missing by default. However, the token table can also be updated by users, who can provide deterministic label values for some of the records.

3.2 MR Matrix: A Materialization of the CRF Model

The probability distribution F over all possible extractions is stored in the MR matrix [1], which is a materialization of the factor tables in the CRF model for all the tokens in $\mathcal{D}$. More specifically, each token x_t in $\mathcal{D}$ is associated with a factor table $\phi[y_t, y_{t-1} \mid x_t]$ in the CRF model, which represents the correlations between the token x_t, the label y_t and the previous label y_{t-1}. The factor table $\phi[y_t, y_{t-1} \mid x_t]$ is computed using the weighted sum of the features activated by the token x_t in the CRF model:

$$\phi[y_t, y_{t-1} \mid x_t] = \sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, x_t).$$

where the features are real-valued functions, as described in Section 2.3. There are two ways to store the MR matrix. The first is to use the following schema, where {token,label,prevLabel} is the primary key:

MR (token, label, prevLabel, score)

An example of MR matrix with the above schema is shown in Figure 1(c) [2]. The second way is to store the factor table $\phi[y_t, y_{t-1} \mid x_t]$ for each token x_t as an array data type, where the array contains a set of scores sorted by {prevLabel, label}. This is a more compact representation, and can lead to better memory locality characteristics. In addition, with the array data type, we do not have to explicitly store the values of prevLabel and label, we can simply look up the score by index. For example, if we want to fetch the score for prevLabel=5 and label=3, then we look up the $(5 \times |Y| + 3)$th cell in the array. The MR matrix schema with the array data type is the following:

MR (token, score ARRAY[])

[1] The name MR matrix is borrowed from the CRF Java implementation [22].

[2] 'DIGIT' in the MR matrix is a generalization of numbers.

3.3 Possible World Semantics

A possible world of this probabilistic database is an assignment to all the missing values in the probabilistic label^p attribute in the TOKENTBL. Each possible world also maps to a unique set of segmentations, one for each document in $\mathcal{D}$. The distribution F over the possible worlds is quantified by the MR matrix, representing the CRF model. With the TOKENTBL and the MR matrix, we can fill in the missing label^p values by computing the top-k inference using Viterbi dynamic programming algorithm. As we discuss in the next section, Dynamic programming algorithms can be implemented using recursive SQL queries. Moreover, the marginal distribution over the missing values can also be computed by a dynamic programming algorithm through recursive SQL queries.

4 CRF Inference Queries

Having described the design of a probabilistic database $\mathcal{DB}^p$ supporting the CRF model, this section describes the recursive SQL implementation of the Viterbi algorithm over $\mathcal{DB}^p$. We compare the merits of a declarative SQL implementation vs. an imperative Java implementation of the Viterbi algorithm, and decide on a middle ground that retains a good deal of the declarativeness of SQL, but embeds imperative UDF functions for issues where relational is not a good representation, such as vectors and arrays.

We also describe the SQL implementation of ConstrainedViterbi for constrained top-k inference queries. Finally, we describe briefly how to extend the Viterbi SQL implementation to compute marginal distribution inference.

4.1 Viterbi SQL Implementations

The Viterbi dynamic programming algorithm can be implemented using recursive SQL queries over the incomplete relation TOKENTBL and the model representation in the MR matrix. We compare different Viterbi implementations including (1) an existing Java implementation (2) the SQL implementations ViterbiPerDoc, ViterbiAllDoc with recursive queries and (3) the SQL implementation ViterbiArray with recursive queries and UDF functions over arrays.

4.1.1 ViterbiPerDoc and ViterbiAllDoc

As stated in Section 2.4.1, the Viterbi algorithm computes the top-k segmentation using a dynamic programming data structure – the V matrix. Let us assume that we are computing top-1 segmentation for simplicity. Each cell in $V(i, y)$ stores the score and the path of the top-1 partial segmentation up until position i ending with label y. The V matrix at position 0 is initialized from the MR matrix: $V(0, y) = \phi[y, -1|x_0]$, where -1 denotes that the previous label is NULL. The V matrix at position $i > 0$ is recursively defined from the V matrix at position $i - 1$, by picking the partial segmentation with maximum score: $V(i, y) = \max_{y'}\{V(i-1, y') + \phi[y, y'|x_i]\}$. The next example illustrates the score and the path of the top-1 segmentations stored in the V matrix.

EXAMPLE 2. *We use the address of "Jupiter", a popular jazz bar in downtown Berkeley, "2181 Shattuck Ave. Berkeley CA USA" as an example. Figure 3(a) shows the V matrix computed by the Viterbi algorithm. The first row contains the possible labels and the first column contains the string positions from 0 to 5. The scores are shown in the cells of the V matrix; the path of the top-1 partial segmentations are shown as the edges. For example, the partial segmentation in V(3,'city') consists of three edges V(0,'street num') → V(1,'street name'), V(1,'street name') → V(2,'street name'), and V(2,'street name') → V(3,'city').*

The top-1 segmentation of the document can be computed by following the path from the cell in the V matrix with the maximum score. In this example the path is high-lighted: {V(0,'street num') → V(1,'street name') → V(2,'street name') → V(3,'city') → V(4,'state') → V(5,'country')}.

```
1 CREATE FUNCTION ViterbiPerDoc (int) RETURN VOID AS
2 $$

3 -- compute the top-1 path from V
4 INSERT INTO Ptop1
5 WITH RECURSIVE P(pos,segID,label,prevLabel,score) AS (
6     SELECT * FROM Vtop1 ORDER BY score DESC LIMIT 1
7     UNION ALL
8     SELECT V.* FROM Vtop1 V, P
9     WHERE V.pos = P.pos-1 AND V.label = P.prevLabel
10 ),

11 -- compute the V matrix from mr and tokenTbl
12 Vtop1 AS(
13   WITH RECURSIVE V(pos,segID,label,prevLabel,score) AS (

14     -- compute V(0,y) from mr and tokenTbl
15     SELECT st.pos, st.segID, mr.label, mr.prevLabel, mr.score
16     FROM tokenTbl st, mr
17     WHERE st.docID=$1 AND st.pos=0 AND mr.segID=st.segID
18             AND mr.prevLabel=-1
19     UNION ALL

20     -- compute V(i,y) from V(i-1,y), mr and tokenTbl
21     SELECT start_pos, seg_id, label, prev_label, score
22     FROM (
23         SELECT pos, segID, label, prevLabel, score, RANK()
24             OVER (PARTITION BY pos,label ORDER BY score DESC) AS r
25         FROM (
26             SELECT st.pos, st.segID, mr.label, mr.prev_label,
27                 (mr.score+v.score) as score
28             FROM tokenTbl st, V, mr
29             WHERE st.docID=$1 AND st.pos = v.pos+1
30                AND mr.segID=st.segID AND mr.prevLabel=v.label
31         ) as A
32     ) as B WHERE r=1
33   )SELECT * FROM V)

34 SELECT $1 as docID, pos, segID, label FROM Ptop1
35 $$
36 LANGUAGE SQL;
```

Figure 2: ViterbiPerDoc UDF function takes in one docID at a time and computes the top-1 segmentation.

The basic SQL implementation of any dynamic programming algorithm involves recursion and window aggregation. In the Viterbi algorithm, the window aggregation is the group by followed by sort and top-1. In Figure 2, we show the SQL implementation for the ViterbiPerDoc algorithm in a UDF function. This algorithm processes one docID at a time.

Lines 11 – 33 compute the V matrix in the Viterbi algorithm. Lines 14 – 18 initialize the V matrix at position 0 for all labels. Lines 26 – 30 compute all possible partial segmentations up until position i by joining the portion of the V matrix at position $i - 1$: $V(i - 1)$ and the portion of MR table for the token at position i: $MR(x_i)$ on condition $V(i-1).label = MR(x_i).prevLabel$. The rest of the logic in lines 20 – 33 computes the top-1 partial segmentation $V(i, y_i)$ among all the partial segmentations up until position i with the same y_i value, using the `rank()` (for computing top-1) function with `partition by` (group by) and `order by` (sort).

After the V matrix is computed, lines 3 – 10 reconstruct the top-1 segmentation for a document, by backtracking from the maximum score in the V matrix. Line 6 picks the cell in the V matrix with the maximum score, and lines 8,9 recursively trace back the top-1 segmentation.

The mode of computation for ViterbiPerDoc is to execute the inference for one document at a time. An alternative is to compute the top-k inference for all documents in $\mathcal{D}$ at the same time. We call this the ViterbiAllDoc algorithm. ViterbiPerDoc might incur more

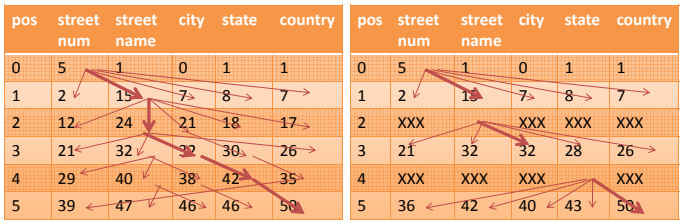


Figure 3: Illustration of the computation of V matrix in the following algorithms: (a) ViterbiPerDoc; (b) ConstrainedViterbi

```
1 SELECT st.pos, st.segID,
2        top1_array(v.score, mr.score) as score
3 FROM tokenTbl st, V, mr
4 WHERE st.docID=$1 AND st.pos = v.pos+1
5        AND mr.segID=st.segID
```

TOP1-ARRAY $(int[2][]arrayV, int[]arrayMR)$
1 $size1 =$ *length of dimension 2 of arrayV*;
2 $size2 =$ *length of dimension 1 of arrayMR*;
3 $result =$ *new* $int[2][size1]$;
4
5 *// compute the top1 for each label over the join result*
6 *// of arrayV and arrayMR on V.label=MR.prevLabel*
7 *// arrayMR is ordered by (prevLabel,label)*
8 **for** $i = 0; i < size2; i$++ **do**
9 *// prevLabel and label in arrayMR*
10 $k = i/size1; k_r = i\%size1;$
11 $newscore = arrayV[0][k] + arrayMR[i];$
12
13 **if** $k == 0 \parallel newscore > result[0][k_r]$ **then**
14 $result[0][k_r] = newscore;$
15 *// record the prevLabel for the top1 for label* k_r
16 $result[1][k_r] = k;$
17 **endif endfor**
18 **return** $result$

Figure 4: ViterbiArray modification in ViterbiPerDoc and the algorithm for UDF function top1-array.

overhead for initializing and tearing down the queries for each document inference; while ViterbiAllDoc might suffer from generating large intermediate table, which cannot be indexed.

The most important change for ViterbiAllDoc from ViterbiPerDoc is omitting the docID= $1 predicate on Lines 17 and 29. In this case, the V matrix becomes three dimensional with docID, position and label.

4.1.2 ViterbiArray

The ViterbiArray algorithm is an optimization of the ViterbiPerDoc algorithm, which uses the second schema of the MR matrix from Section 3.2, and takes advantage of the array data type. As described in Section 3.2, each token x_t in the MR matrix is associated with a one dimensional `score` array of length $|Y| \times |Y|$. The `score` array for a specific token x_t contains values in the factor table $\phi[y_t, y_{t-1} \mid x_t]$, sorted by prevLabel $y_{t=1}$ and label y_t. Correspondingly, the `score` in the V matrix for a specific position i is also stored as an array of $\langle score, prevLabel \rangle$ pairs with length $|Y|$, where prevLabel is used for backtracking the top-1 segmentation.

The ViterbiArray algorithm has similar dynamic programming structure to that of the ViterbiPerDoc algorithm, except that it uses special UDF functions over arrays. Figure 4 shows the SQL statements that replace Line 21 – 32 of ViterbiPerDoc. As we can see, the UDF function TOP1-ARRAY$(V(i-1), MR(x_i))$ in Line 2, replaces the join between array $V(i-1)$ and array $MR(x_i)$ with condition $V(i-1).label = MR(x_i).prevLabel$, and the window aggregation (group-by, sort and top-1 operations) over the result of the join.

```
1 startTok as (
2    -- all tokens without a label which is
3    -- preceded by a token with evidence on label
4    SELECT S1.pos, S1.segID, S1.label, S2.label as prevLabel
5    FROM tokenTbl S1,
6    (
7        SELECT * FROM tokenTbl
8        WHERE docID=$1 and label is not NULL
9    ) as S2
10    WHERE S1.docID=$1 and S1.pos=S2.pos+1 and S1.label is NULL
11    UNION ALL
12
13    -- unlabeled starting token of the document
14    SELECT pos, segID, label, -1 as prevLabel
15    FROM tokenTbl
16    WHERE docID=$1 and pos=0 and label is NULL
17 ),
18 endPos as (
19    -- all tokens with label evidence which is
20    -- preceded by a token without evidence on label
21    SELECT S2.pos
22    FROM tokenTbl S1,
23    (
24        SELECT * FROM tokenTbl
25        WHERE docID=$1 and label is not NULL
26    ) as S2
27    WHERE S1.docID=$1 and S1.pos=S2.pos-1 and S1.label is NULL
28    UNION ALL
29
30    -- unlabeled ending token of the document
31    SELECT pos FROM (
32        SELECT * FROM tokenTbl WHERE docID=$1
33        ORDER BY pos DESC LIMIT 1
34    ) as A WHERE label is NULL
35 )
```

Figure 5: Computing sub-sequences of a document separated by evidence in ConstrainedViterbi algorithm.

Figure 4 also shows the algorithm for the function TOP1-ARRAY, which takes in two arrays: the first is two dimensional `arrayV` $V(i-1)$ and the second is one dimensional `arrayMR` $MR(x_i)$. `arrayV` is a 2 by $|Y|$ array, and `arrayMR` is a $|Y| \times |Y|$ array. Lines 10 – 11, join `arrayV` and `arrayMR` and compute the new scores for the join result tuples. Line 13 – 17 compute the top-1 $\langle score, prevLabel \rangle$ pair for each label y_i. The function returns a new `arrayV` $V(i)$.

The TOP1-ARRAY function in ViterbiArray is much faster than the combination of the join and the window aggregation operations in ViterbiPerDoc for the following reasons:

- the join on $V(i-1).label = MR(x_i).prevLabel$ is replaced by index computation between two arrays $V(i-1).score$ and $MR(x_i).score$;
- the scores in the MR matrix are compactly represented as an array, and the label and prevLabel values are encoded as the array index in ViterbiArray, rather than explicitly stored in ViterbiPerDoc, which greatly improves the memory locality characteristics;
- the computation of the top-1 partial segmentation for each label y_i in ViterbiPerDoc is implemented by group by, sort and top-1, whereas in the TOP1-ARRAY function in ViterbiArray, the group by is replaced by index computation, the sort and the top-1 are replaced by the maintanance of a priority queue.

4.2 Constrained Top-k Inference

A constrained Viterbi algorithm prunes the paths that cannot satisfy the given constraints [16]. One way to do this is to check if the

constraints are satisfied for every step of the path while the path is being generated, and if they are not, the reminder of the path is pruned. The constraints are experessed as values ("evidence") for one or more labels in the CRF sequence model. As such, the label constraints not only improve the performance (through pruning), but also increase the accuracy of the inference. This design follows a "pay-as-you-go" approach [15] in which the system exploits user-provided evidence to improve its performance and accuracy.

Evidence for a label partitions the CRF sequence model into two conditionally independent sub-sequences: one that starts from the beginning of the CRF sequence model up until (but not including) the label with evidence, and the other from the label with evidence until the end of the model. Similarly, multiple label evidence partitions the CRF sequence model into multiple sub-sequences, each separated by one or more labels with evidence. The process of checking a constraint involves identifying the required sub-sequence pattern (expressed as starting and ending positions for each sub-sequence) and performing the top-k inference over them. We can further reduce the overhead of constraint checking by materializing the sub-sequence pattern and letting the database prune paths that do not satisfy the pattern.

Figure 5 shows the SQL to compute the start and end positions of the sub-sequences defined by the evidence in each document. Start tokens ($startTok$) of the sub-sequences include the starting token of the document (unless the start token has label evidence) (Lines 13 – 16), and tokens with no label evidence that are preceeded by a token which has label evidence (Lines 2 – 10). End tokens ($endPos$) of the sub-sequences include the ending token of the document (unless the end token has label evidence) (Lines 30 – 34), and the tokens with label evidence that are preceded by a token without label evidence (Lines 19 – 27).

EXAMPLE 3. *Consider the example* TOKENTBL *in Figure 3(b) with label evidence for tokens at position* 2 *and* 4. *Thus, we can cross out all the cells in* $V(3)$ *where* $y \neq$ *'street name' and in* $V(5)$ *where* $y \neq$ *'city' as shown in Figure 3(b). The start tokens of the sub-sequences are tokens at positions* 0, 3 *and* 5. *The end positions of the sub-sequences are positions* 2, 4 *and* 5. *As we can see by counting the edges in Figures 3(a) and (b), we are effectively computing far fewer paths in ConstrainedViterbi than in the ViterbiPerDoc algorithm.*

Thus, with an increasing number of evidence labels, not only does the performance of inference improve, but so does the accuracy.

4.3 Marginal Inference

The marginal inference computes the marginal distribution over missing label^p values in TOKENTBL. Suppose we want to compute the marginal inference of the label y_t of a single token x_t in $\mathbf{x}$. We first need to compute a forward variable α_t and a backward variable β_t, each of which is a vector of size $|Y|$. The forward and backward variables are defined as:

$$\alpha_t(y_t) = \sum_{y_{t-1} \in Y} f_t(y_t, y_{t-1}, x_t) \cdot \alpha_{t-1}(y_{t-1}), \quad (5)$$

$$\beta_t(y_t) = \sum_{y_{t+1} \in Y} f_{t+1}(y_{t+1}, y_t, x_{t+1}) \cdot \beta_{t+1}(y_{t+1}). \quad (6)$$

with initialization $\alpha_0(y_0) = f_1(y_0, -1, x_0)$ and $\beta_T(y_T) = 1$. As we can see that the α_t and β_t can be computed by similar recursive query as the SQL implementation of Viterbi. The difference is that the scores are multiplied instead of summed and the aggregation is summation instead of top-k.

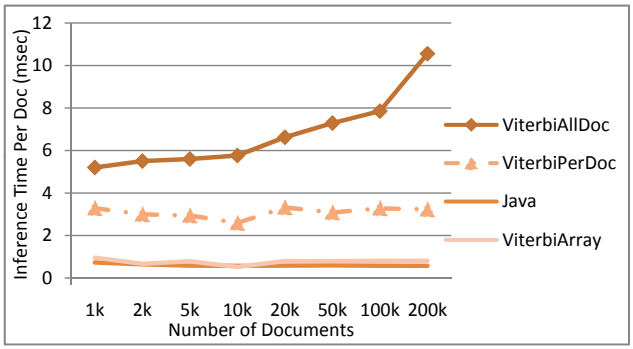


Figure 6: Average inference time (msec) for a single document for different implementations of the Viterbi algorithm.

dataset	ViterbiAllDoc	ViterbiPerDoc	ViterbiArray	Java
address	10.5 msec	3.2 msec	0.8 msec	0.5 msec
bib	1760.1 msec	175.1 msec	6.2 msec	16.2 msec

Figure 7: Average inference time per document (msec) for different Viterbi implementations on address and bib dataset.

With the result of the α_t and β_t values, the marginal distribution of $P(y_t \mid \mathbf{x})$ can be computed as:

$$P(y_t \mid \mathbf{x}) \propto \alpha_t(y_t) \cdot \beta_t(y_t) \quad (7)$$

4.4 Experimental Results

We implemented a probabilistic database supporting the CRF model, inference and relational queries over Postgres8.4 development version. The reason we use the development version is that it supports recursive queries. The Java implementation of the CRF model learning and inference is from the CRF open source project [22]. We conducted our experiments on a 2.4 GHz Intel Pentium 4 Linux system with 1GB RAM. All experiments are run 3 times and take the average running time.

We use two datasets in the evaluation. The first dataset is a set of over $200,000$ address strings we extracted from the yellow book. The average number of tokens in the address strings is 6.8, and 8 labels are used to tag this dataset. The second dataset is a set of more than $18,000$ bibliography entries prepared by R.H. Thomason [27]. The average number of tokens in the bibliography strings is 37.8, and 27 labels are used to tag this dataset.

We ran the three top-k inference implementations in SQL: ViterbiAllDoc, ViterbiPerDoc and ViterbiArray, and the Java implementation on the address dataset. Figure 6 shows the average inference time per document (IPD) in msec for different Viterbi implementations with respect to the increasing number of documents on the x-axis, from 1k to 200k.

The results show that basic SQL implementations of the Viterbi dynamic programming algorithm (ViterbiAllDoc and ViterbiPerDoc) are 6 to 20 times more expensive than the hand-tuned Java implementation. On the other hand, the use of the array data type in ViterbiArray improves the memory characteristics of the program. It demonstrates comparable performance to the Java implementation. The average IPD is 0.57 msec for Java, and 0.81 for ViterbiArray. The graph also shows that the ViterbiPerDoc, ViterbiArray and Java implementations are scalable with the number of documents; while the ViterbiAllDoc implementation is not, because it process all the documents at the same time, thus generating larger intermediate tables as more documents are processed.

We also ran the Viterbi implementations over the bib dataset. Figure 7 compares the IPD numbers in this case. The ViterbiAllDoc and ViterbiPerdoc implementations performs worse on the bib dataset, which has longer documents and more labels. This is again because of their poor memory locality characteristics. Note that the ViterbiArray implementation is more efficient than the Java implementation on the bib dataset. There are two main reasons for this: (1) ViterbiArray uses the materialized MR matrix, which in contrast, needs to be computed on the fly in Java; and (2) the ViterbiArray implementation uses an efficient join between two arrays and has good memory locality characteristics, which is especially evident with a large number of labels.

5 Selection and Top-k Inference

So far we have described a probabilistic database $\mathcal{DB}^p$ that supports CRF model storage and Viterbi inference queries, achieving competitive performance results compared to the inference engine in the Java CRF package. This efficient in-database CRF modeling and inference opens up opportunities to multiplex inference queries with relational operators, and opportunities for optimizing such queries. Eventually, the database optimizer should be taught to optimize such queries based on cost.

In this section, we explore queries that combine top-k inference with selection. To optimize such queries, we would like to push down the selection conditions into the top-k inference algorithm. However, such an optimization is complicated by the fact that the Viterbi algorithm is recursive. We discuss three types of selection conditions: 1) condition on token text; 2) condition on token position/text and label; 3) condition on token label.

5.1 Condition on Token Text

The first type of selection condition is one on the token text. For example, the following query:

```
SELECT *
FROM (  SELECT docID,pos,token,top-1(label)  (I)
        FROM TokenTbl )
WHERE token='York'
```

returns all occurrences of the 'York' token in $\mathcal{D}$, and their labels in the top-1 segmentation as computed by the Viterbi algorithm.

First, instead of computing the top-1 inference for all documents in $\mathcal{D}$, we only need to compute inference on the documents that contain 'York'. In the Java implementation, such a selection can be performed either by scanning through all the documents, or by building an additional inverted index over $\mathcal{D}$. In the SQL implementations, since all tokens in $\mathcal{D}$ are stored in TOKENTBL, we can perform this selection directly over the TOKENTBL efficiently.

Second, if label evidence exists for some tokens in $\mathcal{D}$, instead of computing the top-1 segmentation for each document selected from the first step, we can compute the top-1 inference only on the one sub-sequence of each document that contains the desired token (e.g., 'York'). Given label evidence, the computation of subsequences in a document d is described in Section 4.2. If the desired token already has label evidence, then top-k inference is not necessary.

EXAMPLE 4. *Suppose the document in Figure 3(b) contains the word 'York' in position* 2*. Because of the conditional independence properties of the CRF model, the sub-sequence with position* $[1, 2]$ *is conditionally independent of the rest of the document given evidence on position* 3*. Thus, in order to compute the label for 'York' in the top-1 segmentation, we only need to compute the inference over the sub-sequence* $[1, 2]$ *conditioned on the evidence in position* 3*.*

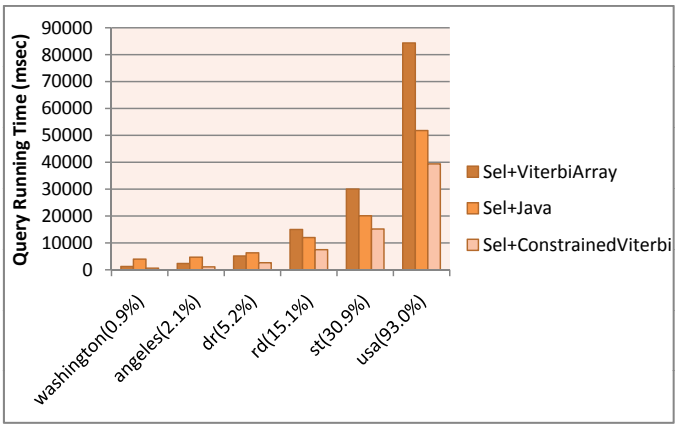


Figure 8: Total top-1 inference time (msec) for 100,000 address strings with selection condition on token text with different selectivities(x-axis).

Figure 8 compares the running time of the query (I) for the three different implementations: Java, ViterbiArray, and ConstrainedViterbi. The first implementation is in Java, where each document in $\mathcal{D}$ is scanned and the inference is computed only on those which contain the token x in the selection condition. For ViterbiArray, the inference is performed over documents containing the token x after applying the selection condition on the TOKENTBL. Neither Java nor ViterbiArray use evidence on label^p. Finally, the ConstrainedViterbi is performed only over the sub-sequences of the document containing x. For ConstrainedViterbi we assume that 10% of the labels have evidence.

The experiment is run over $100,000$ address strings in the address dataset, with different tokens in the selection condition. Different token conditions have different selectivities, which are listed next to the tokens on the x-axis. For example, the token 'washington' occurred in 0.9% of the documents in $\mathcal{D}$. As we can see in Figure 8, with high selectivity the SQL implementations outperform the Java implementation, because SQL uses the index on the tokens. Because the inference using ViterbiArray is, on average, 1.5 times slower than Java, as the selectivity becomes lower, the benefit of indexing reduces and Java performs better than the ViterbiArray. The ConstrainedViterbi always performs better than the other two, because of the presence of the 10% label evidence. The label evidence reduces the inference computation from a full document to a sub-sequence of a document.

5.2 Condition on Position and Label

The second type of condition is one on both the position and the label attribute. One practical application for such queries is when different types of strings are mixed together in one corpus, and by selecting on the label of the first token of the string, we try to pick out all the strings belonging to one type. For example, an address string corpus includes both residential addresses and business PO box addresses. By selecting all the documents with 'P.O. Box' as the label of the first token, we can pick out all the business PO box addresses. This is performed by the following query:

```
SELECT docID
FROM (  SELECT docID,pos,token,top-1(label)  (II)
        FROM TokenTbl )
WHERE label='P.O. Box' and pos=0
```

Unlike a condition on token text, a condition on label is not selective, because most label values in TOKENTBL are missing, and

we have to inference over all the missing labels. A naive way to compute this type of query is to first compute the top-k inference for each document, then check if the selection condition is satisfied.

One possible optimization is to push down the selection condition on a {position,label} pair (e.g. {0,'P.O. Box'}) into the Viterbi algorithm, to make the computation of top-k inference with this type of selection conditions more efficient. In this section, we first describe such a selection-aware Viterbi algorithm – SelViterbi. Then we describe a generalized selection-aware top-k inference algorithm.

5.2.1 Selection-Aware Viterbi

Suppose that the selection condition is {pos= i,label= y'}: i.e., the label of token at position i is y' in the top-1 segmentation of a returned document. In terms of the Viterbi dynamic programming algorithm, this condition is satisfied if and only if one of the top-1 segmentation paths traces back to cell $V(i, y')$ in the V matrix.

The intuition behind the selection-aware Viterbi algorithm is that, for a document d, if none of the top-1 partial segmentations in $V(j)$ traces back to cell $V(i, y')$, then we can conclude that this document does not satisfy the condition $\{i, y'\}$. We call position j a *pruning position* for condition $\{i, y'\}$ in document d. A pruning position for condition $\{i, y'\}$ in a document d is the smallest position in d, where for all possible labels y, there does not exist a partial top-1 segmentation in $V(j)$ that traces back to cell $V(i, y')$.

EXAMPLE 5. *In Figure 3(a), position* 1 *is the pruning position for condition (0, 'street name'), (0, 'city'), (0, 'state'), (0, 'country'), but not for (0, 'street num'). There happens to be no pruning position for (0,'street num') in this example because the top-1 segmentation has label 'street num' for position* 0.

Given the definition of a pruning position, now we describe the algorithm and data structure to efficiently check for a pruning position for condition $\{i, y'\}$ during the Viterbi dynamic programming algorithm, and stop the inference early if the pruning position j is found.

The SQL statements in Figure 9 show how the recursive part of the Viterbi algorithm to compute the V matrix is modified in the SelViterbi algorithm. The new array `filter` is a one dimensional array with size $|Y|$. At a particular iteration j of the recursive query, the cell `filter`[y] stores the information whether $V(j, y)$ traces back to $V(i, y')$ or not. In other words, if `filter`[y]=1 then the partial top-1 segmentation in $V(j, y)$ satisfies the condition $\{i, y'\}$; otherwise the partial segmentation does not have label y' at position i. The recursion stops when the `filter` array becomes a zero array and $pos > i$, which means that none of the partial segmentations at position j satisfy the condition $\{i, y'\}$.

The `filter` array is initialized and updated by the UDF function SEL-ARRAY, shown in Figure 9. Lines 5 – 8 initialize the filter, assigning 1 to the y'th cell in the `filter` array in ith iteration. Lines 9 – 16 update the filter, assigning 1 to the yth cell of the newFilter, if the prevLabel in arrayV[y] is marked 1 in the old `filter` array. The code assigns 0 to the yth cell of the newFilter, if the prevLabel in arrayV[y] is -1 (i.e. starting of the document) or is marked 0 in the old `filter` array.

EXAMPLE 6. *Using the example in Figure 3, suppose we have the condition* {1,*'street num'*}*: return all documents with 'street number' as the label of the second token in the top-1 segmentation. The filter value of the first iteration is* $[0, 0, 0, 0, 0]$. *In the second iteration, the cell in the newFilter corresponding to 'street number', is initialized to* 1. *Thus the filter value at the end of the second iteration is* $[1, 0, 0, 0, 0]$. *In the third iteration, there is no label y with*

```
1 SELECT st.pos, st.segID,
2       topk_array(v.score, mr.score) as score,
3       sel_arry(i,y',v.filter,v.score,v.pos) as filter
4 FROM tokenTbl st, V, mr
5 WHERE st.docID=$1 AND st.pos = v.pos+1 AND
6       mr.segID=st.segID AND (pos<=i or nonzero(v.filter))
```

SEL-ARRAY $(i, y', int[]filter, int[2][]arrayV, pos)$

```
1   size1 = length of filter array;
2   size2 = length of dimension 2 of arrayV;
3   newFilter = new int[size1];
4
5   //initialize the filter array
6   if pos == i then
7     newFilter[y'] = 1;
8   endif
9   // update the filter array
10  for i = 0; i < size2; i++ do
11    // get the previous label
12    prevLabel = arrayV[1][i];
13    if prevLabel == -1||filter[prevLabel]! = 1 then
14      newFilter[i] = 0;
15    else newFilter[i] = 1;
16    endif
17  endfor
18  return newFilter
```

Figure 9: The SelViterbi algorithm uses the sel-array to initialize and update the filter array. A pruning position $j > i$ is found when the filter array contains all zeros.

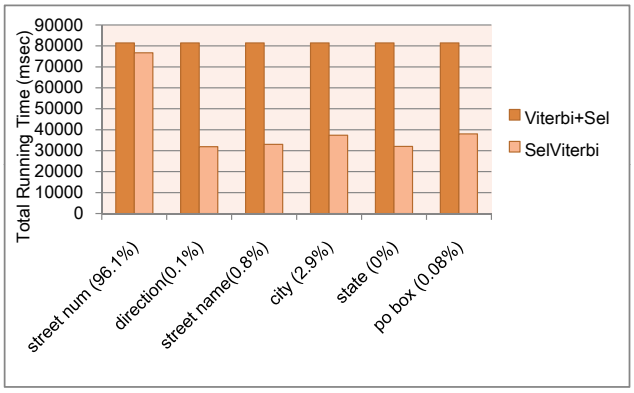


Figure 10: Total top-1 inference time (msec) for 100,000 address strings with selection condition on label (x-axis) at position 0**.**

prevLabel 'street num', no cell in the newFilter is assigned 1. *The value of the filter at the end of the third iteration is* $[0, 0, 0, 0, 0]$. *Thus, we can stop the dynamic programming algorithm and conclude that the document does not satisfy the condition.*

The SelViterbi algorithm can be used for conditions on any (position, label) pair. However, because the Viterbi algorithm computes the top-k segmentation from the start to the end of a document, the early stopping optimization is most effective for conditions with small positions.

Figure 10 shows the running time to compute query (II) with two different SQL implementations: Viterbi+Sel and SelViterbi. Viterbi+Sel computes top-k segmentations using ViterbiArray first, then performs the selection on top of the result. The query (II) is first run over 100,000 address strings in address dataset, with condition on position 0 with different labels. Each label condition has different selectivities at position 0, which are marked on the x-axis. As we can see, 'street num' appears as the first label in 92.1% of the address strings. With such low selectivity, SelViterbi

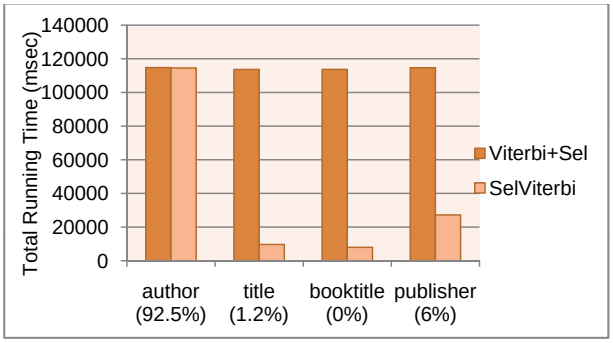


Figure 11: Total top-1 inference time (msec) for 18,000 bibliography entries with selection condition on label (x-axis) at position 0.

still performs slightly better than Viterbi+Sel, which shows that the overhead of checking the pruning position is quite low. For all other label conditions, the SelViterbi performs 2 to 3 times better than Viterbi+Sel.

We conduct the same experiment on the bib dataset with 18,000 bib entries. Figure 11 shows that with longer documents, SelViterbi achieves better speedup because of the early stopping optimization. For the condition of label 'title' at position 0, SelViterbi has more than 10 times speedup compared to Viterbi+Sel.

5.2.2 Generalized Selection Aware Top-k Inference

As we have mentioned, SelViterbi is most efficient with conditions on one of the starting positions and its label. On the other hand, the Viterbi algorithm can be rewritten to compute the top-k segmentation from the end to the start of a document:

$$V(i,y) = \begin{cases} \max_{y'}(V(i+1,y') \\ \quad + \sum_{k=1}^{K} \lambda_k f_k(y', y, x_{i+1})), & if \quad -1 \leqslant i < T \\ 0, & if \quad i = T. \end{cases} \tag{8}$$

where the pruning position optimization is most effective for condition on one of the last few positions.

Furthermore, the dynamic programming algorithm to compute the top-k segmentations for a document can be redefined to start from any position p_0 in the document. Unlike the Viterbi algorithm, this variant of the dynamic programming algorithm expands the intermediate result of the partial segmentations in two directions, rather than just one direction: left to right in Viterbi.

To implement this algorithm, we define a new V matrix which contains four dimensions:
$V(i, y1, j, y2)$, where each cell contains the top-k partial segmentation between position i and position j, where $y1$ is assigned to position i and $y2$ is assigned to position j.

$$V(i,y1,j,y2) = \begin{cases} \max_{y1',y2'}(V(i+1,y1',j-1,y2') \\ \quad + \sum_{k=1}^{K} \lambda_k f_k(y1, y1', x_i) \\ \quad + \sum_{k=1}^{K} \lambda_k f_k(y2', y2, y_{i+1})), \\ \qquad\qquad if - 1 \leqslant i < j \leqslant T \\ 0, \qquad\qquad if i = j = p_0 \end{cases} \tag{9}$$

In order to take advantage of the constraint placed by the selection condition, the starting position p_0 of the inference algorithm should be from the position in the selection condition. With multiple selection conditions, the algorithm should start at the position where the condition is the most restrictive.

Based on the position in the selection condition, an optimizer should decide whether to use SelViterbi, the backward SelViterbi in Formula 8, or the generalized selection aware top-k algorithm in Formula 9.

5.3 Condition on Label

The last type of selection condition is one on the label only, in other words, to pick out all the tokens with a particular label in the top-1 segmentations of the documents in $\mathcal{D}$. Since the label can appear anywhere in any document in $\mathcal{D}$, in most cases, the most efficient query plan for such queries is to compute the top-k inference first and then apply the condition on the result.

However, for highly selective label conditions, if we have a simple classifier, which can pick out the strings with the label with high recall and high selectivity, then we can first apply the classifier to filter out most of the document that does not contain the label, and then apply the CRF model to improve the precision of the inference result. The simple classifiers can be string based (e.g. matching of a regular expression or a small dictionary). We note that unlike the rest of our discussion above, the idea here is a heuristic – it may be of use in practice, but it may sacrifice fidelity with respect to a full Viterbi computation.

For example, to perform data cleaning or data visualization, we need to select all addresses with a certain type of road (e.g. 'rue', 'boulevard', 'lane').

```
SELECT docID, pos, token
FROM (  SELECT docID,pos,token,top-1(label) (III)
        FROM TokenTbl)
WHERE label='Boulevard'
```

For the above query, we can use a dictionary classifier to pick out documents that contain 'blvd' or 'boulevard' or 'blvd.'. Then we can apply the Viterbi algorithm over the resulting documents, and finally applying the selection condition on the inference results. For example, if 5% of the address strings in the corpus $\mathcal{D}$ are on a boulevard, then we can achieve 20 times speedup. However, we may lose some strings that contain the 'boulevard' label that do not contain any word in the dictionary. Such a query plan is especially useful for queries that can trade-off recall for performance improvement.

6 Join with Top-K inference

In this section, we consider how to compute and optimize queries with both top-k inference and join operators. There are two general techniques for optimizing such queries: pushing down join conditions into top-k inference and conditional evaluation. We discuss three types of join queries: (1) self-join over the same model, (2) join between two models over the same document, and (3) join between two models over two different document corpus.

6.1 Self-Join over Single Model

The first type of join is a self-join over the same TOKENTBL and CRF model, to pick out all the documents in $\mathcal{D}$ whose top-1 segmentation $\mathbf{y}$ matches a particular "follow by" pattern. Such queries are very useful for debugging purposes. For example, in most address strings, a street address should not be followed by a street number, but an apartment number. Thus, for data cleaning, the developer may want to find all the documents, whose top-1 segmentation starts with a 'street name' label and followed by a 'street num' label with at most 2 tokens separation.

Let us generalize this query to a join query between the token at position i with label $label1$ and a token with label $label2$ at a position $> i$, with a follow-by join condition $\preceq_n$, where N is the maximum token distance between the two base tokens.

$$\sigma_{<i,label1>}(top1(\text{TOKENTBL}, CRF1)) \bowtie_{\preceq_n} \sigma_{<label2>}(top1(\text{TOKENTBL}, CRF1)) \tag{IV}$$

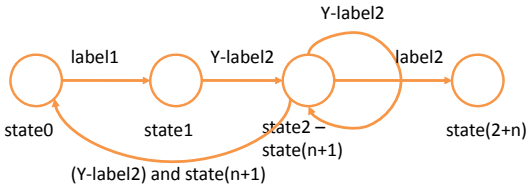


Figure 12: The simple state machine for matching the follow by join condition $\{i, label1, n, label2\}$.

The naive way of computing this query is to compute the top-1 segmentation of all the documents in $\mathcal{D}$, and perform the join on top of the inference results. One possible optimization, as in the case of SelViterbi, is to push the 'follow by' join condition into the Viterbi algorithm. We call this algorithm JoinViterbi.

A 'follow by' join condition can be described by four parameters: $\{i, label1, n, label2\}$, where in the top-1 segmentation $\mathbf{y}^*$ of a document d, the token at position i has label1, which is followed by maximum n tokens with labels in Y except label2, and in the end a token with label2. The condition can be satisfied if and only if the top-1 segmentation $\mathbf{y}^*$ contains the following pattern starts at position i: (label1)(Y-label2)$\{0, n\}$(label2). The pattern represents exactly the 'follow by' condition $\{i, label1, n, label2\}$.

The intuition behind the JoinViterbi algorithm is that for a document d at position j, if none of the top-1 partial segmentations in $V(j)$ is either currently matching or has matched the 'follow by' pattern, then we can conclude that this document does not satisfy the join condition $\{i, label1, n, label2\}$. We call position j a pruning position for $\{i, label1, n, label2\}$, with a similar definition to the pruning position in SelViterbi. A pruning position for a 'follow by' condition $\{i, label1, n, label2\}$ in a document d, is the smallest position in d, where for all possible labels $y \in Y$, there does not exist a partial top-1 segmentation in $V(j)$, which either matches the join condition pattern, or is currently matching the pattern.

EXAMPLE 7. *In Figure 3(a), position 1 is the pruning position for condition (0,'street name',0,'street num'), because no partial segmentation at position 1 trace back to V(0,'street name'). Position 3 is the pruning position for join condition (0,'street num',1,'city'), because all partial segmentations at position* 3 *fails the matching of the join pattern.*

Next we describe the JOIN-ARRAY function, which is used in the JoinViterbi algorithm to detect the pruning position and to trigger early stopping the Viterbi dynamic programming algorithm.

In the JOIN-ARRAY algorithm, we still use a `filter` array with length $|Y|$, with each cell representing a label in Y. The value in each cell of the `filter` array, stores the state in matching the follow by pattern. For example, $\{i, label1, n, label2\}$ condition is represented by the pattern (label1)(Y-label2)$\{0, n\}$(label2).

A segmentation matching this pattern can be in one of the states: $\{0, ..., (2+n)\}$. State 0 represents either nothing is matched yet or the segmentation failed matching. State 1 represents that the current partial segmentation matched (label1) pattern. State $\{2, ..., (1+n)\}$ represent the current partial segmentation is matching (label1)(Y-label1-label2)$\{0, n\}$ pattern, with state 2 corresponds to 1 (Y-label2) label and state $n+1$ corresponds to n (Y-label2) labels. And state $2+n$ represent the final state where the full pattern (label1)(Y-label2)$\{0, n\}$(label2) is matched. As shown in Figure 12, the above states form a simple state machine representing the 'follow by' pattern. The edges in the state machine show four types of transition patterns:

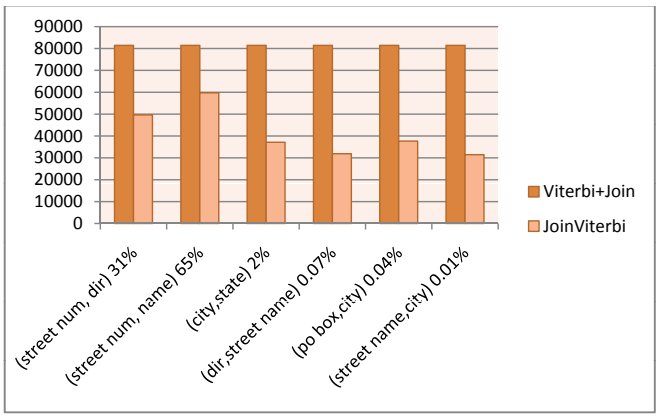


Figure 13: Total inference time (msec) for 100,000 address strings with top-1 inference with different follow by join condition.

1. state0 → state1 by matching label1;
2. state1 → state2, state2 → state3,... state(n) → state(n+1) by matching any labels in Y that is not label2;
3. state2,...,state(n+1) → state3 by matching label2;
4. state(n+1) → state0 by matching any label in Y that is not label2.

Figure 14 shows the modification to the Viterbi recursion in the JoinViterbi algorithm, and the `join-array` algorithm which initializes and updates the filter array to check for pruning positions. Lines 5 – 8 are the initialization of the filter: `filter`[label1] is assigned to state 1 at positions i. Lines 9 – 26 update the filter array at position j, based on the state of the partial segmentation up until position $j-1$ in $filter[prevLabel]$ and the label i of the current position j. There are five cases in the updating loop from Line 10 to 24. The first case (Lines 13 – 14) initializes the states in `filter`, the rest of the four cases correspond to the four types of transition edges in Figure 12 we described earlier.

Figure 13 shows the running time to compute query (IV) with two different SQL implementations: Viterbi+Join and JoinViterbi. Viterbi+Join computes the top-1 segmentation using ViterbiArray first then performs the join on top of the result. Query (IV) is run over 100,000 address strings in address dataset, with 'follow by' join conditions with different label1 at start position 0, and followed by a label with $0-4$ token distance. label1 and label2 and the selectivity of the join conditions are marked on the x-axis of the figure.

As we can see, with high selectivity join conditions JoinViterbi performs approximately 2 times better than Viterbi+Join. As the selectivity becomes lower, the speed up decreases, but even a condition with 65% selectivity, JoinViterbi can still achieve a 25% speed up over Viterbi+Join. Similar to the results with SelViterbi, the speed up of the JoinViterbi algorithm is more a significant over document corpus with longer documents.

6.2 Join between Two Models over Same Document

The second type of join is one between the top-1 segmentation of two different CRF models over the same document. Suppose two different CRF models extract different fields from the same document, and this type of join query can be used with follow by condition to combine the related fields into one record.

$$\sigma_{<label1>}(top1(\text{TOKENTBL}, CRF1)) \bowtie_{\preceq_n} \sigma_{<label2>}(top1(\text{TOKENTBL}, CRF2)) \quad \text{(V)}$$

```
1 SELECT st.pos, st.segID,
2       topk_array(v.score, mr.score) as score,
3       join_array(i,y1,n,y2,v.filter,v.score,v.pos) as filter
4 FROM tokenTbl st, V, mr
5 WHERE st.docID=$1 AND st.pos = v.pos+1 AND
6       mr.segID=st.segID AND (pos<=i or nonzero(v.filter))
```

JOIN-ARRAY $(i, y1, n, y2, int[]filter, int[2][]arrayV, pos)$

```
1  size1 = length of filter array;
2  size2 = length of dimension 2 of arrayV;
3  newFilter = new int[size1];
4
5  //initialize the filter array
6  if pos == i then
7     newFilter[y1] = 1;
8  endif
9  // update the filter array
10 for i = 0; i < size2; i++ do
11    // get the previous label
12    prevLabel = arrayV[1][i];
13    if prevLabel == −1||filter[prevLabel] == 0 then
14       newFilter[i] = 0;
15    else if filter[prevLabel] < (1 + n)&i! = y2 then
16       newFilter[i] = filter[i] + 1;
17    else if filter[prevLabel] <= (i + n)&i == y2 then
18       newFilter[i] = 2 + n;
19    else if filter[prevLabel] == 2 + n then
20       newFilter[i] = 2 + n;
21    else if filter[prevLabe] == 1 + n&i! = y2 then
22       newFilter[i] = 0;
23    endif
24 endfor
25 return newFilter
```

Figure 14: JoinViterbi algorithm uses JOIN-ARRAY to initialize and update filter array. A pruning position is reached when filter array contains all zeros.

For example, we have a email document corpus and two CRF models are trained to extract person names and telephone numbers. We can use the above query to return person-telephone pairs that appear within 5 tokens in the same document.

The above query can be optimized by computing the top-1 inference of CRF2 on TOKENTBL conditionally, only when the top-1 inference of CRF1 on TOKENTBL generates a tuple with label1. This technique is called "conditional evaluation" [21, 24].

Suppose two different CRF models extract the same set of fields from the same document, and this type of join query can be used with an equality join condition to only return when the two models agree with each other.

$$top1(\text{TOKENTBL}, CRF1) \bowtie_{docID=docID, pos1=pos2, label1=label2} top1(\text{TOKENTBL}, CRF2) \quad \text{(VI)}$$

This type of query can be used to build ensemble models, which combine multiple weak (e.g. low-precision) models to construct a strong (e.g. high-precision) model.

If the query is to ensemble the inference results of multiple models on all labels, the most efficient query plan is to compute the top-1 segmentation for each document using both CRF1 and CRF2, and perform the join. If the query is to ensemble the inference results of multiple models over a particular label, then we can use the "conditional evaluation" to perform the inference with CRF2 based on the results of the inference with CRF1.

6.3 Join between Two Models over Different Documents

So far, we have limited our discussion to extraction models over the same set of documents. However, in real life, the same entity can be extracted from multiple different document corpus. For example, profiles of the same person can be extracted from multiple social networks, such as Facebook, MySpace, and LinkedIn, etc. Different sources have different set of extracted attributes. Suppose we want to join the extracted people profiles from Facebook and MySpace into one table, joined on the address fields.

The first step to compute this query is to create a "pivot" view on top of the address TOKENTBL, where each row in the view represents an address with fields 'street name', 'city', etc. For example, Figure 1(d) shows the pivot view for two possible segmentations of the address string. The second step is to compute the following query with a 'pivot' operator:

$$pivot(top1(\text{TOKENTBL}_1, CRF1)) \bowtie_{\sigma*} pivot(top1(\text{TOKENTBL}_2, CRF2)) \quad \text{(VII)}$$

where the join condition $\sigma*$ is that 'street num', 'street name', 'city', 'state', 'country' fields in the base views being the same. The pivot operator is not supported by Postgres8.4-Dev, however, there are different ways to implement it and some database vendors support it (e.g. SQL Server).

Conditional evaluation can also be used to optimize this query.

7 Related Work

Information extraction (IE) from text has received a lot of attention both the database and the Machine Learning (ML) communities [17, 6, 13, 16, 18, 25]. (See [1, 9] for recent tutorials.) The vast majority of works in ML focus on improving extraction accuracy using state-of-the-art probabilistic techniques, including different variants of HMM [25] and CRF [26, 17, 16] models. Probabilistic IE models provide high-accuracy extraction results and a principled way to reason about the uncertainty of the IE process; still, such tools are typically imperative and special-purpose, and are not targeted at the declarative management and processing of large-scale data sets.

While the approaches presented so far are an indicative sample of the basic building blocks of a real-world IE system, a lot of effort has recently been devoted on exploring *frameworks* that manage the state of the IE process, and provide support to easily specify, optimize and execute IE applications. Most promising approaches in this category are those which are based on *declarative* specifications of the IE tasks at hand. SYSTEMT [21], is an algebraic, rule-based IE system, developed by IBM Research. Through the use of an SQL variant, IE programs (rules) can be declaratively expressed. The system uses database-inspired cost-based optimization techniques to execute the resulting workflows efficiently. Shen et al. [24] also consider optimizing declarative IE programs. Their work extends DataLog with an embedded IE predicate, ans also proposes specialized pattern indexing schemes. These earlier efforts in declarative IE did not consider supporting a declarative interface over state-of-the-art probablistic models for IE; furthermore, they did not address the inherent uncertainty of the IE process.

At the same time, the ML community has also been moving in the direction of declarative IE. Eisner et. al. [10] adopts a pure first-order logic perspective by extending Prolog to provide support for dynamic programming. No database style cost-based optimization is considered, but machine-learning optimizations are developed, which the user can opt to include in her applications. Markov Logic [19] represent a very powerful graphical model that marries the declarativeness of first-order logic with the popular Markov Network model. Although no large-scale IE system has been built to date based on this mechanism, [19, 20] successfully exemplify how their model can handle the task of citation matching, through

the use of a handful of intuitive first-order logic rules. In contrast to the above systems, this framework natively supports uncertainty modeling, as it is based on a sophisticated machine learning model, similar to our own approach.

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 [3, 5, 7, 4, 8, 23, 2, 28]. This work extends the relational model with probabilistic information captured at the level of individual *tuple existence* (i.e., a tuple may or may not exist in the DB) [5, 7, 12, 23] or individual *tuple-value uncertainty* (i.e., an attribute value in a tuple follows a probabilistic distribution) [3, 4, 2]. The Trio [4] and MayBMS [2] efforts, try to adopt both types of uncertainty, with Trio focusing on promoting *data lineage* as a first-class citizen in PDBSs and MayBMS aiming at more efficient tuple-level uncertainty representations through effective relational table decompositions. Recent PDBS efforts like BAYESSTORE [28] and the work of Sen and Deshpande [23] represent probabilistic models (based on Bayesian networks) as first-class citizens in relational database, and support in-database queries and reasoning over the the model. The issue of offering database support for managing IE through state-of-the-art probabilistic models has not been addressed in existing PDBSs. Closer to our work Gupta and Sarawagi [14] give tools for storing coarse approximations of a CRF model inside a simple PDBS supporting limited forms of tuple- and attribute-level uncertainty. Instead, our work aims to support the full expressive power of CRF models and inference operations as a first-class PDBS citizen.

8 Conclusion

In this work we develop a probabilistic database approach to CRF-based IE models. This includes implementing Viterbi inference as SQL queries, as well as integrating and optimizing top-k inference queries with relational operators. Our work here incorporates a specific family of probabilistic models – linear-chain CRF models – as a first-class citizen in a probabilistic database. The algorithms and optimizations in this paper take advantage of this specific model's structure to improve the efficiency of inference and relational operations.

Our previous work on BAYESSTORE [28] provides a much more general approach to capturing first-order Bayesian networks as first-class citizens in a probabilistic database. Given the broad class of models support by BAYESSTORE, it did not focus on model-specific techniques like Viterbi. In future work we hope to address this shortcoming in the large, by enriching BAYESSTORE with an extensible framework that can support and optimize itself to a wide variety of graphical models: general or specialized, directed or undirected, generative or discriminative. This entails a number of challenges, including extending the query processor to support inference and relational queries over different models, extending the query optimizer to perform cost-based optimization based on the query and the model structure, and designing an extensible API for specifying new models.

9 Acknowledgements

Thanks for Rahul Gupta from IIT Bombay, who provided help with using the Java CRF package [22].

10 References

[1] E. Agichtein and S. Sarawagi. Scalable Information Extraction and Integration (tutorial). In *KDD*, 2006.

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

[3] D. Barbará, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Trans. on Knowl. and Data Eng.*, 4(5):487–502, 1992.

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

[5] R. Cavallo and M. Pittarelli. The Theory of Probabilistic Databases. In *VLDB*, 1987.

[6] A. Culotta and A. McCallum. Confidence Estimation for Information Extraction. In *HLT-NAACL*, 2004.

[7] N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In *VLDB*, 2004.

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

[9] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing Information Extraction: State of the Art and Research Directions (tutorial). In *SIGMOD*, 2006.

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

[11] G. D. Forney. The Viterbi Algorithm. *IEEE*, 61(3):268–278, March 1973.

[12] N. Fuhr and T. Rolleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Transactions on Information Systems*, 15:32–66, 1997.

[13] T. Grenager, D. Klein, and C. D. Manning. Unsupervised Learning of Field Segmentation Models for Information Extraction. In *ACL*, 2005.

[14] R. Gupta and S. Sarawagi. Curating Probabilistic Databases from Information Extraction Models. In *VLDB*, 2006.

[15] S. Jeffery, M. Franklin, and A. Halevy. Pay-as-you-go User Feedback for Dataspace Systems. In *SIGMOD*, 2008.

[16] T. Kristjansson, A. Culotta, P. Viola, and A. McCallum. Interactive Information Extraction with Constrained Conditional Random Fields. In *AAAI'04: Proceedings of the 19th National Conference on Artificial Intelligence*, 2004.

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

[18] L. Peshkin and A. Pfeffer. Bayesian Information Extraction Network. In *Proceedings of the IJCAI*, 2003.

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

[20] H. Poon and P. Domingos. Joint Unsupervised Coreference Resolution with Markov Logic. In *Proc. of ACL*, 2008.

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

[22] S. Sarawagi. CRF Open Source Project.`http://crf.sourceforge.net/`.

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

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

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

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

[27] R. Thomason. `http://www.eecs.umich.edu/~rthomaso/bibs/bigbibs.html`.

[28] D. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models. In *VLDB'08: Proceedings of the 34th International Conference on Very Large Data Bases*, 2008.