Probabilistic Declarative Information Extraction

Daisy Zhe Wang #[1], Eirinaios Michelakis #[2], Michael J. Franklin #[3], Minos Garofalakis *[4], Joseph M. Hellerstein #[5]
#*EECS, University of California, Berkeley, USA*
{ [1]daisyw, [2]ireneos, [3]franklin, [5]hellerstein }@cs.berkeley.edu
**Technical University of Crete, Greece*
[4]minos@acm.org

***Abstract*— Unstructured text represents a large fraction of the world's data. It often contains snippets of structured information (e.g., people's names and zip codes). 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 probabilistic databases for querying the output of IE. In this paper, we make the first step to merge these two directions, without loss of statistical robustness, by implementing a state-of-the-art statistical IE model – Conditional Random Fields (CRF) – in the setting of a Probabilistic Database that treats statistical models as first-class data objects. We show that the Viterbi algorithm for CRF inference can be specified declaratively in recursive SQL. We also show the performance benefits relative to a standalone open-source Viterbi implementation. This work opens up the optimization opportunities for queries involving both inference and relational operators over IE models.**

I. Introduction

The field of database management has traditionally focused on *structured* data, however, there has been significant interest 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 [1], [2], [3]. Second, IE has been a major motivating application for the recent 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 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 declarative 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 (CRF) [10] 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 storing the probabilistic output of the CRF in a probabilistic database [11]. However, they used a PDBS model that supported *tuple-* and *attribute-level uncertainty* with restrictive dependency structures. This limitation enabled them to capture only a coarse approximation of the CRF distribution model inside the PDBS, when complex dependency structures exist.

In this paper we show how to bridge the gap between CRF-based IE and probabilistic databases, preserving the fidelity of the CRF distribution. Our technique is based on the following observations:

- *Relational Representation of Inputs:* CRF can be naturally modeled as first-class data in a relational database, in the spirit of recent PDBS like BayesStore [9] and the work of Sen and Deshpande [7]. Similarly, text data can be captured relationally via the inverted file representation commonly used in information retrieval.
- *Declarative Viterbi Inference:* Given tabular representations of CRF model parameters and input text, the central algorithm for CRF – Viterbi inference [12] – can be elegantly expressed as a standard recursive SQL query for dynamic programming.

Together, these results in a unified and efficient approach for implementing a probabilistic database supporting CRF and its inference operator in PostgreSQL DBMS. We also show the performance benefits relative to a standalone open-source Viterbi implementation. Importantly, our approach not only correctly performs CRF-based IE on input text, it also maintains the probability distributions inherent in CRF to enable standard *possible worlds* semantics for answering queries over the model, which is part of future work.

II. Background

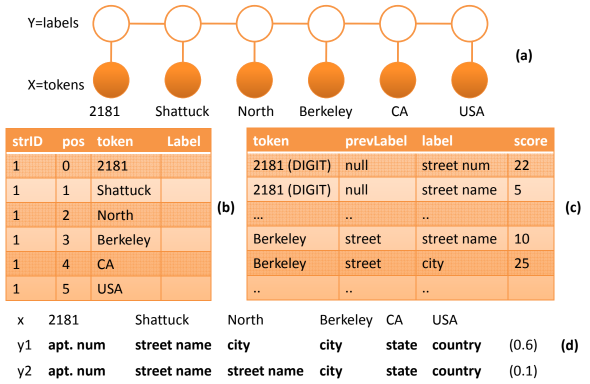
This section covers the background on probabilistic databases, the CRF model, and the top-k inference operation over a CRF model, in the context of information extraction.

A. 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 $\mathcal{R}$.

B. Conditional Random Fields (CRF)

The declarative IE system we build and evaluate is based on the linear-chain Conditional Random Field (CRF) [13], which is a leading probabilistic model for solving IE tasks.



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	**apt. num**	**street name**	**city**	**city**	**state**	**country**	(0.6)
y2	**apt. num**	**street name**	**street name**	**city**	**state**	**country**	(0.1)

Fig. 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 an example IE task called *field segmentation*, in which a text-string is regarded as a sequence of pertinent fields, and the goal is to tag each token in a text-string with one of the field labels.

Example 1: Fig. 1(a) shows a CRF model instantiated 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 token in $\mathbf{x}$ into one of the field labels in Y. □

The following definition [10], [13] 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 variables, $\Lambda = \{\lambda_k\} \in R^K$ 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 a normalization function. □

Two possible feature functions for Example 1 are:

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

A CRF model represents the probability distribution over all possible segmentations of a text-string *d*. Fig. 1(d) shows two possible segmentations of *d* and their probabilities.

C. Top-k Inference on CRF Model

The top-k inference is the most frequently used inference operation over the CRF model. Top-k inference determines the *segmentations* with the top-k highest probabilities given a token sequence $\mathbf{x}$ from a text-string *d*.

The *Viterbi* dynamic programming algorithm [12] 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.

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

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 text-string and $|Y|$ is the number of labels.

III. CRF MODELS IN A PROBABILISTIC DATABASE

In this section, we describe a probabilistic database $\mathcal{DB}^p$=<$\mathcal{R}$, F>, based on [9] that can support rich probabilistic IE models, such as CRF by (1) storing text-strings 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 using a factor table.

A. Token Table: The Incomplete Relation

The token table TOKENTBL is an incomplete relation R, which stores text-strings as relations in a database, in a manner akin to the inverted files commonly used in information retrieval. As shown in Fig. 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. A TOKENTBL has the following schema:

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

The TOKENTBL contains one probabilistic attribute – label^p, which can contain missing values, whose probability distribution can be computed from the CRF model. The deterministic attributes of the TOKENTBL are populated by parsing the input text-strings $\mathcal{D}$, with label values marked as missing by default.

B. MR Matrix: A Materialization of the CRF Model

The probability distribution F over all possible extractions is stored in the MR matrix (the name MR matrix is borrowed from [14]), 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 from the weighted sum of the features activated by 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 II-B. 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 Fig. 1(c). 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:

MR (token, score ARRAY[])

IV. Top-k Inference on CRF Model

This section describes the recursive SQL implementation of the Viterbi algorithm. 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.

A. 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 ViterbiPerStr, ViterbiAllStr with recursive queries and (3) the SQL implementation ViterbiArray with recursive queries and UDF functions over arrays.

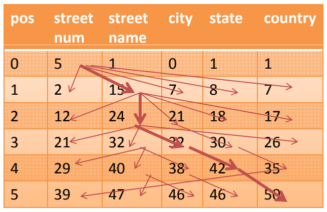
ViterbiPerStr and ViterbiAllStr: As stated in Section II-C, the Viterbi algorithm computes the top-k segmentation using a dynamic programming data structure – the V matrix. Let us assume that we are computing top-1 segmentation for simplicity. Each cell in $V(i, y)$ stores the score and the path of the 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. Fig. 3 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 text-string 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 ViterbiPerStr (int) RETURN VOID AS
2 $$
3 -- compute the top-1 path from V
4 INSERT INTO Ptop1
5 WITH RECURSIVE P(pos,segID,label,prevLabel,score) AS (
6     SELECT * FROM Vtop1 ORDER BY score DESC LIMIT 1
7     UNION ALL
8     SELECT V.* FROM Vtop1 V, P
9     WHERE V.pos = P.pos-1 AND V.label = P.prevLabel
10 ),
11 -- compute the V matrix from mr and tokenTbl
12 Vtop1 AS(
13   WITH RECURSIVE V(pos,segID,label,prevLabel,score) AS (
14     -- compute V(0,y) from mr and tokenTbl
15     SELECT st.pos, st.segID, mr.label, mr.prevLabel, mr.score
16     FROM tokenTbl st, mr
17     WHERE st.strID=$1 AND st.pos=0 AND mr.segID=st.segID
18          AND mr.prevLabel=-1
19     UNION ALL
20     -- compute V(i,y) from V(i-1,y), mr and tokenTbl
21     SELECT start_pos, seg_id, label, prev_label, score
22     FROM (
23       SELECT pos, segID, label, prevLabel, score, RANK()
24            OVER (PARTITION BY pos,label ORDER BY score DESC) AS r
25       FROM (
26           SELECT st.pos, st.segID, mr.label, mr.prev_label,
27                  (mr.score+v.score) as score
28           FROM tokenTbl st, V, mr
29           WHERE st.strID=$1 AND st.pos = v.pos+1
30                 AND mr.segID=st.segID AND mr.prevLabel=v.label
31       ) as A
32     ) as B WHERE r=1
33 )SELECT * FROM V)
34 SELECT $1 as strID, pos, segID, label FROM Ptop1
35 $$
36 LANGUAGE SQL;
```

Fig. 2. ViterbiPerStr UDF function in SQL takes in one strID at a time and computes the top-1 segmentation.



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	32	30	26
4	29	40	38	42	35
5	39	47	46	46	50

Fig. 3. Illustration of computing V matrix in ViterbiPerStr algorithm.

The basic SQL implementation of any dynamic programming algorithm involves recursion and window aggregation. In the Viterbi algorithm, the window aggregation is `group by` followed by `sort` and `top-1`. In Fig. 2, we show the SQL implementation for the ViterbiPerStr algorithm in a UDF function in SQL. ViterbiPerStr processes one strID at a time, whereas an alternative is to compute the top-k inference for all text-strings in $\mathcal{D}$ at the same time, which we call the ViterbiAllStr algorithm. However, ViterbiAllStr suffers from generating a large intermediate table that cannot be indexed.

ViterbiArray: The ViterbiArray algorithm is an optimization of the ViterbiPerStr algorithm, which uses the second schema of the MR matrix from Section III-B, and takes advantage of the array data type. 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.

Fig. 4 shows the SQL statements in ViterbiArray that replace Line 21 – 32 of ViterbiPerStr, where the UDF function

```
SELECT st.pos, st.segID,
      top1_array(v.score, mr.score) as score
FROM tokenTbl st, V, mr
WHERE st.strID=$1 AND st.pos = v.pos+1
      AND mr.segID=st.segID
```

Fig. 4. ViterbiArray modification in ViterbiPerStr.

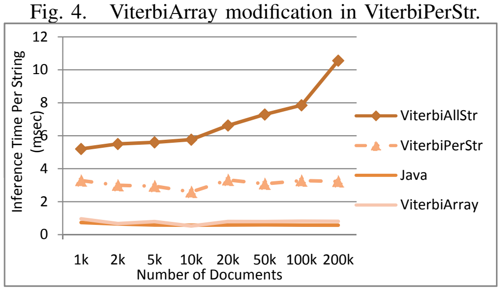


Fig. 5. Average inference time (msec) for a single text-string for different implementations of the Viterbi algorithm.

TOP1-ARRAY$(V(i-1), MR(x_i))$ is used to replace the join between array $V(i-1)$ and array $MR(x_i)$, and the window aggregation (group-by, sort and top-1 operations) over the result of the join.

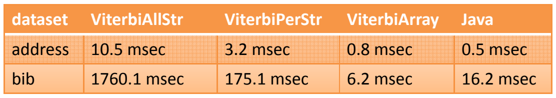
The ViterbiArray algorithm is much faster than the ViterbiPerStr algorithm, because (1) the join on $V(i-1).label = MR(x_i).prevLabel$ is replaced by index computation between two arrays $V(i-1).score$ and $MR(x_i).score$; (2) the scores in the MR matrix are compactly represented as an array, which greatly improves the memory locality characteristics; (3) the computation of the group by, sort and top-1 computation in ViterbiPerStr is replaced by index computation and the maintenance of a priority queue.

B. Experimental Results

We implemented probabilistic declarative IE system supporting CRF and top-k inference over PostgreSQL 8.4 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 [14]. We conducted our experiments on a 2.4 GHz Intel Pentium 4 Linux system with 1GB RAM.

We use two datasets in the evaluation. The first dataset is a set of over $200k$ address strings we extracted from the yellow pages. 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 $18k$ bibliography entries prepared by R.H. Thomason [15]. 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: ViterbiAllStr, ViterbiPerStr and ViterbiArray, and the Java implementation on the address dataset.

Fig. 5 shows the average inference time per text-string (IPS) in msec for different Viterbi implementations with increasing number of text-strings on the x-axis, from $1k$ to $200k$. The results show that ViterbiAllStr is not scalable with the number of text-strings, and ViterbiPerStr is 6 times more expensive than the hand-tuned Java implementation. On the other hand, the use of the array data type in ViterbiArray demonstrates comparable performance to the Java implementation. The average IPS is 0.57 msec for Java, and 0.81 for ViterbiArray.



dataset	ViterbiAllStr	ViterbiPerStr	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

Fig. 6. Average inference time per text-string (msec) for different Viterbi implementations on address and bib dataset.

Fig. 6 compares the IPS numbers over two datasets. The ViterbiArray is more efficient than the Java implementation on the bibliography dataset. There are two main reasons for this: (1) ViterbiArray uses the materialized MR matrix, which needs to be computed on the fly in the Java implementation; and (2) ViterbiArray uses an efficient join between arrays with good memory locality characteristics, whose benefit is especially evident with a large set of labels.

V. Conclusion

In this work, we show the design, implementation, and efficiency of a unified probabilistic database system that supports the CRF IE model and its top-k inference operator. This is the first step towards a declarative IE system that provides a probabilistic framework for performing IE tasks, and querying the outputs of those tasks. This work opens up optimization opportunities for queries with both inference and relational operators over IE models.

Acknowledgment

Thanks for Rahul Gupta from IIT Bombay, who provided help with using the Java CRF package [14].

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

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

[12] G. D. Forney, “The Viterbi Algorithm,” *IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.

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

[14] S. Sarawagi, “CRF Open Source Project.`http://crf.sourceforge.net/`.”

[15] R. Thomason, “`http://www.eecs.umich.edu/r̃thomaso/bibs/bigbibs.html`.”