2010 12th International Asia-Pacific Web Conference

Recording How-Provenance on Probabilistic Databases

Ming Gao #1, Xiangnan He *2, Cheqing Jin *2, Xiaoling Wang *2, Aoying Zhou *2

# *Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China*
1mgao@fudan.edu.cn

* *Software Engineering Institute, East China Normal University, Shanghai, China*
2{xnhe, cqjin, xlwang, ayzhou}@sei.ecnu.edu.cn

Abstract— **Tracking data provenance (or lineage) has become increasingly important in many large-scale applications, and a few methods have been proposed to record data provenance recently. However, most of previous works mainly focus on deterministic databases except Trio style lineage that aims at probabilistic databases, which is much more challenging because of the exponential growth of possible world instances and dependence among intermediate tuples. This paper proposes an approach, named PHP-tree, to model *how-provenance* upon probabilistic databases. we also show how to evaluate probability based on a PHP-tree. Compared with Trio style lineage, our approach is independent of intermediate results and can calculate the probability both cases of restricted and complete propagation of data provenance. Detailed experimental results show the effectiveness, efficiency and scalability of our proposed model.**

I. Introduction

Aiming at indicating how the data is processed and propagated from source to its current form, the data provenance (or lineage) is employed intensively in many fields, inclusive of data quality, audit trail, replication recipes, data citation, etc. Nowadays, a few ways are proposed to describe data provenance over relational databases. Representative works consist of *where-* and *why-* provenance [1], *how-* provenance and B[X] provenance [2], [3], Trio style lineage [4], and lineage over data warehouse [5].

Management on uncertain and probabilistic databases has become an increasing hotter topic in database community [6], [7], [8], since data uncertainty widely exists in a lot of applications, such as data cleaning and integration, information extraction, moving objects and sensor data management, RFID data and so on. In general, there are two kinds of uncertainties, named *existential uncertainty* and *attribute level uncertainty*. Here, *existential uncertainty* is used to describe whether a tuple exists or not, and *attribute level uncertainty* is used to describe imprecise attribute values. More recently, some researchers pay attentions to the provenance upon probabilistic database. The ULDB [4] is the first prototype that can handle provenance over probabilistic databases, where *Trio style lineage* (lineage in short) is proposed to describe the provenance. Sarma et al. [8] studied how to evaluate probabilities for SQL-like queries on ULDB. In their method, probabilities of query results are efficiently calculated based on lineage reserved in advance.

The lineage storage strategy could be treated as restricted propagation since the lineage of the result data item inherits only last data transformation in ULDB. To find the origin of a data item, the representation of lineage can not avoid tracking intermediate tuples, thus we can not directly use the lineage for probability evaluation. Instead, we should expand the lineage of each atom in result data item by lineage formulas, until all variables refer to the base tuples.

TABLE I
The Base Probabilistic Relation R_0

tupleID	color	kid	length	probability
t_1	red	Tom	long	0.6
t_2	yellow	Tom	short	0.8
t_3	red	Tom	short	0.5
t_4	yellow	Mary	short	0.9

TABLE II
R_1 and R_2 Annotated by Lineage

tupleID	lineage	tupleID	lineage
t_{11}	$t_1 \vee t_3$	t_{21}	t_1
t_{12}	t_2	t_{22}	$t_2 \vee t_3$
t_{13}	t_4	t_{23}	t_4

Example 1: Let's consider a tiny table *Candle(color, kid, length, probability)* to record kids' candles information, as shown in Table I. For example, tuple t_3 means that Tom owns one *red*, *short* candle with probability 0.5. Now, our task is to find: *all kids who own red candle(s) and short candle(s) at the same time*. This query can be written as Q_4 that is also based on Q_1, Q_2, and Q_3, as shown below.

$$Q_1 : R_1 = \Pi_{color,kid}(R_0)$$

$$Q_2 : R_2 = \Pi_{kid,length}(R_0)$$

$$Q_3 : R_3 = R_1 \bowtie R_2$$

TABLE III
R_3 Annotated by Lineage and How-Provenance

tupleID	lineage	how-provenance
t_{31}	$t_{11} \vee t_{21}$	$t_1^2 + t_1 t_3$
t_{32}	$t_{11} \vee t_{22}$	$t_1 t_2 + t_1 t_3 + t_2 t_3 + t_3^2$
t_{33}	$t_{12} \vee t_{21}$	$t_1 t_2$
t_{34}	$t_{12} \vee t_{22}$	$t_2^2 + t_2 t_3$
t_{35}	$t_{13} \vee t_{23}$	t_4^2

TABLE IV
R_4 Annotated by Lineage and How-Provenance

tupleID	lineage	how-provenance
t_{41}	t_{32}	$t_3^2 + t_1 t_3 + t_1 t_2 + t_2 t_3$

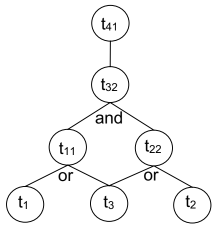


Fig. 1. A Diagram of Lineage for Probability Evaluation

$$Q_4 : R_4 = \Pi_{kid}(\sigma_{color=\text{“}red\text{”}} \text{ and } {}_{length=\text{“}short\text{”}} R_3)$$

Let t_{ij} denote the jth tuple in relation R_i. Tables II-IV show the lineages of all tuples in $R_1 - R_4$. Keep in mind that the "lineage" in this paper always refers to the "Trio-style lineage" [4]. Moreover, Tables III-IV also describe the how-provenance of each tuple in R_3 and R_4, respectively. It is clear to observe that the lineage model records the data evolving information in many relations. For instance, tuple t_{41} comes from t_{32} (in Table IV), which further comes from t_{11} and t_{22} (in Table III). In addition, tuples t_{11} and t_{22} also come from (t_1, t_3), and (t_2, t_3), according to Table II. As a conclusion, to find the whole lineage of tuple t_{41} as Figure 1, it is necessary to visit all five relations listed in Table I-IV, which makes the data exchange very complex and difficult. Contrarily, by using the how-provenance, we can easily find how the tuple t_{41} evolves from the base tuples t_1, t_2, and t_3 in Table IV, which makes the data exchange very clear and convenient.

The difference between lineage and how-provenance may become critical in some realistic applications, and is also implemented in some main projects. For example, Orchestra is a collaborative data exchange prototype that supports peers in a P2P network sharing data and resource with each other. It uses how-provenance to describe how data tuples are evolving among lots of sources. As claimed in [9], the how-provenance used in Orchestra can satisfy two goals: "(i) reconciliation can choose between transactions based on user preferences, and (ii) efficient incremental recomputation of the target data instance and provenance is possible."

As discussed above, it is also necessary to record the how-provenance for uncertain databases. However, how to calculate the probability of a result tuple still remains as a great challenge. For example, how to compute the existential probability of t_{41} if given the how-provenance in advance?

This paper explores a novel approach, named PHP-tree, to cope with this issue. A PHP-tree consists of a tuple header-table and a tree. The contributions of this paper are summarized below.

- We propose a novel approach, named PHP-tree, to approximately describe the how- provenance upon probabilistic databases, avoiding tracking too many intermediate results. Based on PHP-tree, the existential probability of a tuple can be calculated easily.
- Experimental results evaluate the effectiveness, efficiency and scalability of our new approach.

The rest of paper is organized as follows. Section II introduces some preliminary knowledge. Section III describes our PHP-tree approach in detail. A systematic performance study is reported in Section IV. We review the related works in Section V. Finally, we conclude our work briefly and propose some extensions.

II. Preliminary Knowledge

A. How-Provenance

Currently, how-provenance (also called provenance polynomial) mainly focuses on managing deterministic databases. How-provenance not only includes the bags information about involving base tuples, but also includes the times information about involving base tuples, etc. It records information about origin data items and the process of their derivations.

One character of how-provenance is that it is described by base tuples. Let $t.howp$ denote the how-provenance of tuple t. For instance, the how-provenance of t_{41} (Table IV) is $t_3^2 + t_1t_3+t_1t_2+t_2t_3$(denoted as $t_{41}.howp = t_3^2+t_1t_3+t_1t_2+t_2t_3$), indicating that t_{41} can be computed by the self-join of $\{t_3\}$, or join of tuple set $\{t_1, t_3\}$, or $\{t_1, t_2\}$, or $\{t_2, t_3\}$. Furthermore, the coefficient of t_1t_3 (here, the value of the coefficient of t_1t_3 is 1.), indicates that there is only one way to join tuples t_1 and t_3. Compared with other provenances, how-provenance is the strongest one in the aspect of representative capability.

B. How-Provenance upon Probabilistic Databases

Probability and how-provenance in probabilistic databases can be viewed as meta-data of tuples in deterministic databases. Generally, how-provenance can track the process of data propagation and origin of base tuples, as well as the propagation of uncertainty.

For example, the source of uncertainty of t_{41} is t_1, t_2 and t_3, and the propagation of uncertainty is inferred from $t_{41}.howp$. It is also critical to compute the probability of a result tuple based on its how-provenance.

Let $\mathcal{R}$ denote a set of all k base relations, $\mathcal{R} = \{R_1, \cdots, R_k\}$, where $|R_i| = n_i$, $1 \leq i \leq k$. Let $\mathcal{T} = \{t_1, \cdots, t_n\}$ denote all base tuples in relations R_i, $1 \leq i \leq k$, where $n = \sum_{i=1}^{k} n_i$. We also use $Pr(t_i)$ to denote the probability of tuple t_i. In this way, the how-provenance of a tuple t in result relation is defined below.

$$t.howp = \sum_{i=1}^{h}(b_i \prod_{t_j \in S_i} t_j^{n_{i,j}}) \tag{1}$$

, where h is the number of monomials in t.howp. The ith monomial has $b_i \in N$ as its coefficient and is product of base tuples, which consist of a set S_i. In addition, $n_{i,j}$ denotes the power of the tuple t_j in the ith monomial.

We summarize two theorems and three important properties about how-provenance to evaluate the probability .

Property 1: (Independence of Power): Assume $t.howp$ is defined by Equation (1), and

$$t.howp' = \sum_{i=1}^{h}(b_i \prod_{t_j \in S_i} t_j) \tag{2}$$

, then we have $Pr(t.howp) = Pr(t.howp^{'})$, where $Pr(t.howp)$ registers the probability of tuple t based on $t.howp$.

Proof: Each monomial in a how-provenance can be treated as a probabilistic event. So, when a base tuple occurs multiple times in an event, the probability of this event is identical to another event where the same tuple only occurs once. In other words, the power of a base tuple in how-provenance is independent of the probability evaluation. ■

Property 2: (Independence of Coefficient:) Assume $t.howp^{'}$ is defined by Equation (2), and

$$t.howp^{''} = \sum_{i=1}^{h}(\prod_{t_j \in S_i} t_j) \tag{3}$$

, then we have $Pr(t.howp^{''}) = Pr(t.howp^{'})$.

Proof: Each coefficient of a monomial in a how-provenance can be viewed as the occurring times of the same event. So, when an event occurs multiple times, the probability of union of multiple events is equal to the probability that the same event only occurs once. In other words, the coefficient of a monomial in a how-provenance is independent of the probability evaluation. ■

According to Property 1-2, the coefficient of monomials in $t.howp$ and the power of base tuples in the monomial will not influence the final probability calculation. Hence, we can execute probability evaluation based on S_l. In fact, S_l records the existing tuples in lth monomial of $t.howp$. Let $t.RHP$ register the refinement of how-provenance of tuple t, denoted $t.RHP = \{S_1, S_2, \cdots, S_h\}$. For instance, $t_{41}.RHP = \{\{t_3\}, \{t_1, t_2\}, \{t_1, t_3\}, \{t_2, t_3\}\}$. We have $Pr(t.howp) = Pr(t.RHP)$, where $Pr(t.RHP)$ is the probability of tuple t based on $t.RHP$. Next, Theorem 1 indicates how to calculate $Pr(t)$ based on $t.RHP$.

Theorem 1: Let $t.RHP = \{S_1, S_2, \cdots, S_h\}$, then the probability of tuple t is calculated by Equation (4):

$$\begin{aligned} Pr(t) &= \Sigma_{i=1}^{h} Pr(S_i) - \Sigma_{1 \le i,j \le h, i \ne j} Pr(S_i \cup S_j) \\ &+ \Sigma_{1 \le i,j,l \le h, i \ne j \ne l} Pr(S_i \cup S_j \cup S_l) \\ &+ \cdots + (-1)^{h-1} Pr(S_1 \cup S_2 \cup \cdots \cup S_h), \end{aligned} \tag{4}$$

where $S = \cup_{i \in I} S_i$, and I is a subscript subset of $\{1, 2, \cdots, h\}$, $Pr(S) = \prod_{t_j \in S} Pr(t_j)$, $l = 1, 2, \cdots, h$.

Proof: For each $S_i \in t.RHP, i = 1, 2, \cdots, h$, let $t_j \in S_i$, according to the possible worlds model, we can easily obtain $Pr(S) = \prod_{t_j \in S} Pr(t_j)$. Thus, $Pr(t)$ can be easily calculated by addition formula of probability with h events. ■

However, when h is not small, the processing cost will be high because there are $2^h - 1$ monomials in the RHS of Equation (4). According to Theorem 2, we can reduce the h to a small one.

Theorem 2: Let $t.RHP = \{S_1, S_2, \cdots, S_h\}$. If $S_j, S_l \in t.RHP$ and $S_j \subseteq S_l$, we denote $t.RHP^{'} = t.RHP \setminus \{S_l\}$, then we have $Pr(t.RHP) = Pr(t.RHP^{'})$.

Proof: Let $V \subseteq t.RHP$, and $S_j, S_l \notin V$. If $S_j \subseteq S_l$, then probability $Pr(V \cup S_l)$ is equal to $Pr(V \cup S_l \cup S_j)$ because $V \cup S_l = V \cup S_l \cup S_j$ under condition $S_j \subseteq S_l$. In the RHS of Equation (4), the signs of $Pr(V \cup S_l)$ and $Pr(V \cup S_l \cup S_j)$ are opposite, so that both $Pr(V \cup S_l)$ and $Pr(V \cup S_l \cup S_j)$ can be removed from the RHS of Equation (4). Hence, any of set $S = \cup_{i \in I^{'}} S_i$ and $I^{'}$ is a subscript subset of $\{1, 2, \cdots, l-1, l+1, \cdots, h\}$, $Pr(S \cup S_l)$ can be removed form in the RHS Equation (4). So, the probability of t can also be obtained by $Pr(t.RHP^{'})$. ■

Theorem 2 indicates that some information in $t.RHP$ is redundant for probability evaluation. It is clear that $Pr(t)$ remains unchanged after removing the redundant information (also called pruning operation). For best case, Property 3 describes the property of linear complexity about probability evaluation.

Property 3: (Law of Iterative:) Assume $\forall i \ne j$, $S_i \cap S_j = \emptyset$, $\mathcal{Z}_l = \sum_{i=1}^{l}(\prod_{t_j \in S_i} t_j)$ and $Pr(\mathcal{Z}_0) = 0$, where $1 \le i, j, l \le h$, then we have

$$Pr(\mathcal{Z}_l) = Pr(\mathcal{Z}_{l-1}) + Pr(S_l) - Pr(\mathcal{Z}_{l-1})Pr(S_l) \tag{5}$$

and $Pr(t.howp) = Pr(\mathcal{Z}_h)$, where $Pr(S_l) = \prod_{t_j \in S_l} Pr(t_j)$.

Proof: We can iteratively calculate $Pr(\mathcal{Z}_l)$ by Equation (5), we have

$$Pr(\mathcal{Z}_l) = 1 - \prod_{i=1}^{l}(1 - Pr(S_i)) \tag{6}$$

, the RHS of Equation (6) is identical to the probability that at least one of l independent events happens. It also is equal to the probability of adverse event of all l events not occurring, where $1 \le l \le h$. If $l = h$, we have $Pr(\mathcal{Z}_h) = Pr(t.RHP) = 1 - \prod_{i=1}^{h}(1 - Pr(S_i))$. ■

Property 3 indicates, when there are no common tuples between different elements in $t.RHP$, that $Pr(t)$ can be iteratively evaluated with linear time complexity. For example, $t.RHP = \{\{t_1\}, \{t_2, t_3\}, \{t_4\}\}$, the condition of Property 3 is held. So, we can calculate $Pr(t)$ with twice iterations based on $Pr(S_1) = Pr(t_1)$, $Pr(S_2) = Pr(t_2)Pr(t_3)$ and $Pr(S_3) = Pr(t_4)$. But this condition is very rigorous. In most situations, the common subset is not empty between different elements in $t.RHP$, hence the law of iterative is not always correct. So, it is necessary to simplify the process of probability evaluation. In next section, based on a set $t.RHP = \{S_1, S_2, \cdots, S_h\}$, we will introduce the PHP-tree approach to complete it.

III. PHP-tree Approach

This section describes our PHP-tree (pruning how-provenance tree, PHP-tree in short) approach in detail. We introduce the way to maintain a PHP-tree in Section III-A, and show the probability evaluation in Section III-B.

A. Maintaining a PHP-tree

We have shown above that $t_{41}.RHP = \{\{t_3\}, \{t_1, t_2\}, \{t_1, t_3\}, \{t_2, t_3\}\}$. It is easy to deduce the process of data derivation that four combinations of base tuples can be propagated into tuple t_{41}. However, it is still not easy to calculate probability because of complicated dependence among the elements of $t.RHP$. Consequently, we propose a structure, named PHP-tree, to evaluate probability. A PHP-tree consists of a header-table and a tree containing base tuples in $t.RHP$. To make the new structure compact and informative, frequent tuples are arranged close to the "root" node.

- The header-table is in format of ($tupleName$, $leafNode$, $frequency$, $nodeLink$) , here $tupleName$ labels a base tuple, $leafNode$ denotes whether the tuple exists in a leaf node of a branch, where a branch grows in the tree, which is a list of a sorted S_i in $t.RHP$, $frequency$ registers the tuple frequency occurring in $t.RHP$, $nodeLink$ is a pointer pointing to the first node in the PHP-tree carrying the $tupleName$.
- Each node in the tree consists of three fields: $tupleName$, $count$, and $nodeLink$, where $count$ registers the number of branches going by the $tupleName$, and $nodeLink$ links to the next node in the PHP-tree carrying the same $tupleName$, or null if there is none.

Algorithm 1: maintainPHPTree

```
**Input**: $t.RHP = \{S_1, S_2, \cdots, S_h\}$;
**Output**: the PHP-tree of tuple $t$;
1  Scan $t.RHP$ to create a header-table;
2  Create a empty tree with root $F$ labeled as "null";
3  for j=1 to h do
4      Insert($S_j$,F);
5  end
6  for j=1 to M, where M is the size of header-table; do
7      if $t_j.frequency > 1$ then
8          $N = t_j.nodeLink$;
9          while $N \neq null$ do
10             if $N.count > \sum_{N' \in N.children} N'.count$ then
11                 Pruning-rule I;
12             end
13             if $t_j.leafNode = true$ then
14                 Construct a set $V$, which consists of all
                   branches carrying base tuple $t_j$;
15             end
16             $N = N.nodeLink$;
17         end
18         if $|V| > 1$ then
19             Pruning-rule II;
20         end
21     end
22 end
23 return
```

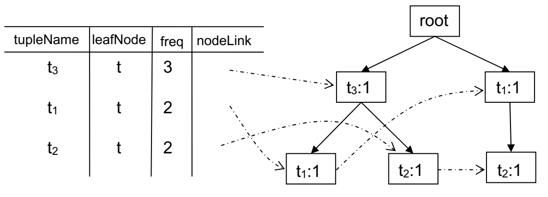
Algorithm `maintainPHPTree` (Algorithm 1) constructs a PHP-tree by scanning $t.RHP$ twice. First, it constructs a header-table containing all base tuples in $t.RHP$ with its frequency(Line 1). Keep in mind that all elements in a header-table are sorted in descending frequency. In the second scan, it constructs a tree with root F labeled "null" based on the header-table(Lines 2-5), here the operation *Insert(S_j, F)* inserts the element S_j in $t.RHP$ into the tree. Subsequently, it runs two pruning rules to make the tree compact (Lines 6-23). If the frequency of tuple t_j in header-table is 1, it do not execute the pruning operations, because t_j impossibly exists in multiple branches(Lines 7-9). Else, it links t_j to the node in the tree by using $t_j.nodeLink$ in header-table(Line 8), and executes *pruning-rule I*(Lines 9-17) and *pruning-rule II*(Lines 18-20) based on constructing set V in advance(Lines 13-15).

Three crucial operations in Algorithm `maintainPHPTree` are described in detail below.

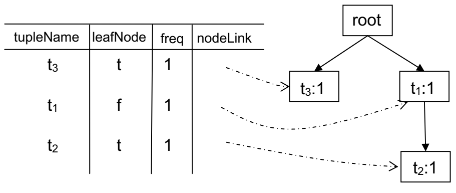
- *Insert(S_j,F)*: Before inserting each element of $t.RHP$ into the tree, we sort each element in S_j by descending frequency of base tuples, and S_j is transformed into a sorted list $(p|P)$, where p is the prefix of S_j and P is the remainder. If F has a child N such that $N.tupleName = p$, then increase $N.count$ by 1; else create a new node N with its $count$ initialized 1 and construct a pointer links to node N via $nodeLink$. If P is nonempty calls *Insert(P, N)*, else updates the $P.leafNode = true$ in the header-table because P is the last tuple in sorted S_j.
- *pruning-rule I*: For any node N in the tree, if $N.count > \sum_{N' \in N.children} N'.count$, then deletes all the descendant of node N and updates the header-table and the tree.
- *pruning-rule II*: Only the leaf node of a branch B appears in another branch, B maybe own the pruning capability. So, if $t_j.leafNode = true$, the algorithm constructs a set V (Lines 13-15), which consists of all branches carrying base tuple t_j. Let branch $b \in V$ and t_j be the leaf node of b, the other branch $B \in V$. If $b \subset B$ meets, then deletes the branch B, and updates the header-table and the tree.

Example 2: Let $t.RHP = \{\{t_3\}, \{t_1, t_2\}, \{t_1, t_3\}, \{t_2, t_3\}\}$, where $S_1 = \{t_3\}, S_2 = \{t_1, t_2\}, S_3 = \{t_1, t_3\}$ and $S_4 = \{t_2, t_3\}$. Initially, we compute the frequency of all tuples in a form $(tupleName : frequency)$(the number after ":" indicates the frequency of tuple $tupleName$ in $t.RHP$). Here, $\{(tupleName : frequency)\} = \{(t_3 : 3), (t_1 : 2), (t_2 : 2)\}$.

- Processing S_1 constructs the first branch of the tree: $< (t_3 : 1) >$ and updates $t_3.leafNode = true$ in the header-table, and processing S_2 constructs the second branch of the tree $< (t_1 : 1), (t_2 : 1) >$ and updates $t_2.leafNode = true$ in the header-table.
- For S_3, since its sorted list $< t_3, t_1 >$ shares a common prefix t_3 with the existing branch $< (t_3 : 1) >$, the count of node $(t_3 : 1)$ increases by 1, and a new node $(t_1 : 1)$ is created and linked as a child of $(t_3 : 2)$ and updates $t_1.leafNode = true$ in the header-table. The processing of S_4 is similar to the S_3. Finally, we can construct a tree as in Figure 2(a)(Lines 1-5 in Algorithm 1).
- Now, we continue to scan the header-table to obtain a PHP-tree by pruning operations. Since $t_3.frequency$ is equal to 3, the algorithm links to the first node in tree by $nodeLink$. Because $(t_3 : 3)$ only has two children $(t_1 : 1)$ and $(t_2 : 1)$, by *pruning rule I*, nodes $(t_1 : 1)$ and $(t_2 : 1)$ can be deleted, the count of



(a)



(b)

Fig. 2. An Example of Constructing PHP-tree

node t_3 update to 1. At the same time, the header-table is updated to $\{(tupleName, leafNode, frequency)\} = \{(t_3, t, 1), (t_1, f, 1), (t_2, t, 1)\}$. Finally, a new PHP-tree is shown in Figure 2(b)(Lines 6-22 in Algorithm 1).

B. Probability Evaluation Based on PHP-tree

For tuple t_{41} in Table IV, $Pr(t_{41})$ can be calculated based on $t_{41}.PHP-tree$ (as Figure 2(b)) by Equation (4), this approach reduce the cost to calculate 3 probabilities. However, on some conditions, the probability evaluation may still be very complex, making us seeking new way to simplify the process of probability evaluation. To reduce the cost, we propose the Algorithm `calProb` (Algorithm 2) to improve the efficiency of probability evaluation.

Algorithm `calProb` (Algorithm 2) illustrates how to simplify probability calculation based on a PHP-tree. First, we group h_0 distinct branches into a collection of disjoint sets by link relationships (Lines 1-17), where h_0 denotes the number of branches in the PHP-tree. In Lines 1-3, we construct h_0 sets V_i. Then we scan the header-table with $t_i.frequency > 1$, and unite two set V_{j_i} and V_{l_i} if they contain a common base tuple t_i(Lines 4-17), where a key operation *unionSet(N)* puts all branches going by node N into a set denoted V_{j_i} and deletes unnecessary sets, here, $V_{j_i} = \cup_{m \in I_i''} V_m$, I_i'' related to base tuple t_i is a subscript subset of $\{1, 2, \cdots, h_0\}$ and $j_i = MIN_{m \in I_i''} m$. After executing Lines 1-17, assume that we group h_0 branches into g disjoint sets, where a disjoint-set data structure maintains a collection $\mathcal{V} = \{V_1, V_2, \cdots, V_g\}$. Subsequently, we calculate probability of each $V_i, 1 \leq i \leq g$ by Equation (4)(Line 18). In this collection, each V_i can be viewed as a probabilistic event, there is no common base tuple between V_i and V_j, $1 \leq i, j \leq g$ and $i \neq j$, so they can be treated as independent events. According to Property 3 in Section II-B, the final probability can be iteratively calculated by multiple probabilities $p_i, 1 \leq i \leq g$(Line 19).

For example, we construct disjoint sets from Figure 2(b) as $V_1 = \{\{t_1, t_2\}\}$ and $V_2 = \{\{t_3\}\}$. Furthermore, we obtain two different probability values $Pr(V_1) = Pr(t_1)Pr(t_2) = 0.48$ and $Pr(V_2) = Pr(t_3) = 0.5$. Finally, we calculate $Pr(t_{41})$ as $0.5 + 0.48 - 0.5 \times 0.48 = 0.74$.

Algorithm 2: calProb

```
   Input: The PHP-tree of tuple t;
   Output: the probability of tuple t;
1  for i=1 to $h_0$, where $h_0$ denotes the number of branches in the input PHP-tree; do
2      Create a set $V_i$,
       $V_i = \{\cup\{t_j\} | t_j$ existing in the $i$th branch$\}$;
3  end
4  for i=1 to $M_0$, where $M_0$ denotes the number of base tuples in the header-table; do
5      if $t_i.frequency > 1$ then
6          $N = t_i.nodeLink$;
7          $unionSet(N) \equiv V_{j_i}$;
8          while $N.nodeLink \neq null$ do
9              $unionSet(N.nodeLink) \equiv V_{l_i}$;
10             if $V_{j_i} \neq V_{l_i}$; then
11                 $V_{j_i}$=$V_{j_i} \cup V_{l_i}$;
12                 Delete $V_{l_i}$;
13             end
14             $N = N.nodeLink$;
15         end
16     end
17 end
18 Compute each $p_i = Pr(V_i)$ by Equation (4), where $1 \leq i \leq |\mathcal{V}|$;
19 $Pr(t) = 1 - \prod_{i=1}^{|\mathcal{V}|}(1 - p_i)$;
20 return Pr(t);
```

Next, we begin to analyze complexity of our approach about probability evaluation. Obviously, under the best situation, there is no intersect between branches in a PHP-tree. In other words, all branches are independent, the complexity of time is $O(g)$, where $g = |\mathcal{V}|$. In the worst case, any of two branches are connected by $nodeLink$. So that, because all branches are in a set, the complexity of time is $O(2^g)$. In the other case, the complexity of time is bounded by $O(g \times 2^{g_0})$, where $g_0 = Max_{1 \leq i \leq g}|V_i|$. For complexity of space, let the total number of base tuples in input PHP-tree be H, then it is $O(H)$.

IV. Experimental Evaluation

A. Experiment Settings

In this section, we evaluate the effectiveness, efficiency and scalability of our novel PHP-tree approach in terms of two measures, the total number of base tuples in a probabilistic database and the complexity of data derivation (represented by the maximum length of branches in PHP-tree and the number of monomials in a how-provenance, respectively). All codes are written in Java and run in the WinXP system with Pentium(R) 2.80GHz CPU and 1G DDR memory. We execute algorithms over following data sets.

- Frequent Item Mining Data Sets: We use two data sets, T10I4D100K(.gz)(SD for short) and T40I10D100K (.gz)(CD for short)[10] , which are generated by the generator from the IBM Almaden Quest research group,

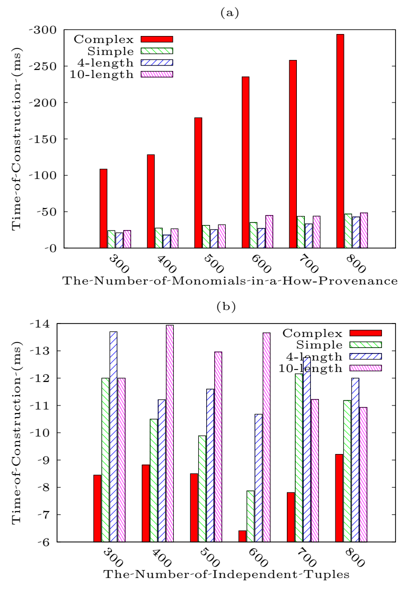


Fig. 3. The Complexity of Constructing a PHP-tree

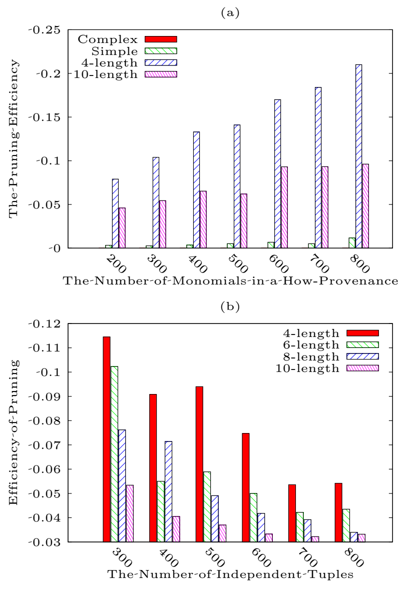


Fig. 4. The Pruning Efficiency over a PHP-tree

downloaded from Frequent Itemset Mining Implementations Repository.

- Synthetic Data Sets: We randomly generate a branch of the PHP-tree with maximum length $l = 4, 6, 8, 10$, and each node of the branch from the set of base tuples with size varying from 100 to 1000. Obviously, varying length of branches in PHP-tree and the number of monomials in how-provenance represent the complexity of data derivation.

B. *Evaluation of the PHP-tree Construction*

We first illustrate the performance of constructing a PHP-tree over both real and synthetic data sets. Specifically, Figure 3(a) illustrates the time cost for constructing PHP-trees over four data sets, SD, CD, 4-length and 10-length's synthetic data. In it, the x-axis registers the number of the monomials in a how-provenance, the y-axis registers the time of PHP-tree's construction. Figure 3(b) illustrates the scalability test on the number of base tuples in probabilistic databases, the x-axis represents the total number of base tuples in the probabilistic databases, and the y-axis is the same to Figure 3(a). We can see that the required time of constructing a PHP-tree is linear about the number of the monomials in a how-provenance, and is almost independent of the size of base tuples' set, which indicates the scalability of our approach against the size of databases.

C. *Pruning Efficiency of PHP-tree*

Subsequently, we start to evaluate the pruning efficiency over different complexity of data derivation and total number of base tuples. The pruning efficiency is shown in Figure 4(a) for complexity of data derivation, and in Figure 4(b) for scalability test on the total number of base tuples in the probabilistic databases, all the x-axes are the same to Figure 3, and the y-axes represent the rate of pruning against $t.RHP$. In the test results, the pruning efficiency increases smoothly with the increasing the number of monomials in a how-provenance. Moreover, the pruning efficiency also becomes higher, when the length of each branch is shorter and the size of database is smaller, which indicates the good scalability in complexity of data derivation, compared with the lower complexity of data derivation.

D. *Evaluation of the Probabilistic Propagation*

We report results of probability evaluation on synthetic data sets by varying length of branches in PHP-tree and the number of monomials in a how provenance. All the y-axes represent the time of probability evaluation, all the x-axes are the same to the Figure 3 as above. Figure 5(a) shows that the complexity of probability evaluation is almost linear relationship to the number of monomials in a how-provenance when $g_0 \leq 5$. Figure 5(b) shows that the time of probability evaluation has not clear tendency when the size of database varies. In other words, probability evaluation has good scalability on the number of base tuples in probabilistic databases.

V. Related Work

The paper by Wang and Madnick [11] firstly studies data provenance in heterogeneous database systems. Subsequently, many researchers define the data provenance (or lineage, pedigree) [12], [13], [1]. Buneman et al [14] define data provenance in the context database systems as the description of the origins of data and the process by which is arrived at in database. Lanter [12] refers to lineage of derived products in geographic information systems (GIS) as information that describes materials and transformations applied to derive the data. Simmhan et al [13] define data provenance to be information that helps determine the derivation history of a data product, starting from its original sources. Y. Cui and J. Widom [5] propose data lineage in data warehouse. And

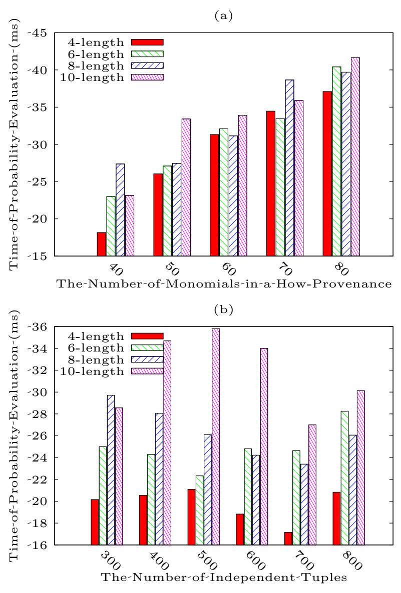


Fig. 5. The Complexity of Probability Evaluation

Buneman et al [1] define the where- and why-provenance that is used to track the processing of derivations of data objects. The paper by Green et al [2] propose the concept of provenance semiring, which data provenance is annotated by polynomials of semiring in relational databases and XML data, and have discussed the containment and equivalence of conjunctive queries on relations annotated by elements of a commutative semiring. Thus, techniques of complete propagation on diversity data provenance do not utilize to probabilistic databases.

Though there has been extensive works on probabilistic databases [6], [7], [8], but each makes simplifying assumptions for getting around the problem of high query evaluation complexity that lessens their applicability. Generally, the possible worlds model is commonly used for representing the uncertainty [15]. The project of Trio by Stanford develops the ULDB database system which integrates uncertainty and lineage into the relational databases. Many works are completed based on this project. The papers by Benjelloun [16] researches the model for uncertainty and lineage of x-tuples. A. D. Sarma have discussed the function dependency of uncertain relations [8] and the optimized algorithms of probability calculation which is related to the intermediate tuples. However, comparing with our work, previous works are dependent of the intermediate results and is impracticable for complete propagation of diversity of data provenances.

VI. Conclusions and Future Challenges

At current, many applications records data provenance for tracking the origin of data item and its derivations. For some probabilistic databases, which data item is annotated by how-provenance, we maintain the approximate how-provenance, called PHP-tree, for probability evaluation upon the probabilistic databases. To summarize, our approach has great advantages. First, it separates the data and probability evaluation in query processing, rather than simultaneously computes them. The query engine can select the best query plan and query optimization techniques to execute the query. Second, doing this averts the possible worlds to model the probabilistic databases, but the computing result is identical to the possible world semantics. Finally, the storage strategy of how-provenance is complete propagation, it leads to probability evaluation is independent of the intermediate dependent results.

Probabilistic databases need to handle more complex probabilistic models, rather than independent tuples discussed in this paper. So, a first evident extension to our work is a complex probabilistic model, because containment correlations between tuples or attribute values naturally occur in practice, such as mutually exclusive and implication relationship.

In large-scale applications, sophisticated queries often depend on multiple data sources instead of single one, that leads to the uncertainty maybe take place in different data sources. So, the other obvious extension to our work is more abundant data schemas, rather than the relational data model.

References

[1] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *ICDT*, 2001, pp. 316–330.
[2] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2007, pp. 31–40.
[3] T. J. Green, "Containment of conjunctive queries on annotated relations," in *ICDT*, 2009, pp. 296–309.
[4] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom, "Uldbs: Databases with uncertainty and lineage," in *VLDB*, 2006, pp. 953–964.
[5] Y. Cui and J. Widom, "Practical lineage tracing in data warehouses," in *ICDE*, 2000, pp. 367–378.
[6] L. Antova, T. Jansen, C. Koch, and D. Olteanu, "Fast and simple relational processing of uncertain data," in *ICDE*, 2008, pp. 983–992.
[7] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004, pp. 864–875.
[8] A. D. Sarma, M. Theobald, and J. Widom, "Exploiting lineage for confidence computation in uncertain and probabilistic databases," in *ICDE*, 2008, pp. 1023–1032.
[9] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen, "Orchestra: facilitating collaborative data sharing," in *SIGMOD Conference*, 2007, pp. 1131–1133.
[10] (2002) The website of frequent itemset mining dataset repository. [Online]. Available: http://fimi.cs.helsinki.fi/data/
[11] Y. R. Wang and S. E. Madnick, "A polygen model for heterogeneous database systems: The source tagging perspective," in *VLDB*, 1990, pp. 519–538.
[12] D. P. Lanter, "Design of a lineage-based meta-data base for gis," *Cartography and Geographic Information Systems*, vol. 18, pp. 255–261, 1991.
[13] Y. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Record*, vol. 34, no. 3, pp. 31–36, 2005.
[14] P. Buneman, S. Khanna, and W. C. Tan, "Data provenance: Some basic issues," in *FSTTCS*, 2000, pp. 87–93.
[15] D. Suciu and N. N. Dalvi, "Foundations of probabilistic answers to queries," in *SIGMOD Conference*, 2005, p. 963.
[16] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom, "Databases with uncertainty and lineage," *VLDB J.*, vol. 17, no. 2, pp. 243–264, 2008.