Optimizing Approximations of DNF Query Lineage in Probabilistic XML

Asma Souihli [#1], Pierre Senellart [#2]

[#]*Institut Mines–Télécom; Télécom ParisTech; CNRS LTCI*
75634 Paris Cedex 13, France
[1]asma.souihli@telecom-paristech.fr
[2]pierre.senellart@telecom-paristech.fr

***Abstract*—Probabilistic XML is a probabilistic model for uncertain tree-structured data, with applications to data integration, information extraction, or uncertain version control. We explore in this work efficient algorithms for evaluating tree-pattern queries with joins over probabilistic XML or, more specifically, for listing the answers to a query along with their computed or approximated probability. The approach relies on, first, producing the lineage query by evaluating it over the probabilistic XML document, and, second, looking for an optimal strategy to compute the probability of the lineage formula. This latter part relies on a query-optimizer–like approach: exploring different evaluation plans for different parts of the formula and estimating the cost of each plan, using a cost model for the various evaluation algorithms. We demonstrate the efficiency of this approach on datasets used in previous research on probabilistic XML querying, as well as on synthetic data. We also compare the performance of our query engine with EvalDP [1], Trio [2], and MayBMS/SPROUT [3].**

I. Introduction

Uncertainty comes along with data generated by imprecise automatic tasks such as data extraction and integration, data mining, or natural language processing. A measure of this uncertainty may be induced from the trustworthiness of the resources, the quality of the data mapping procedure, etc. Often, information is described in a semistructured manner because representation by means of a hierarchy of nodes is natural, especially when the source (e.g., XML or HTML) is already in this form. One possible way, among the most natural, to represent this uncertainty is through probabilistic databases, and probabilistic XML [4] in particular.

Probabilistic documents or *p-documents* [5], [6] are a general representation system for probabilistic XML, based on probabilistic XML trees with ordinary and distributional nodes. The latter define the process of generating a random XML instance following the specified distribution at the level of each node. The model is a compact and complete representation of a probabilistic space of documents (i.e., a finite set of possible worlds, each with a particular probability).

Probabilistic documents have been used in various applications [4], such as uncertain data integration [7], XML warehousing, uncertain version control systems [8], or Web information extraction. We do not deal here with obtaining probabilistic documents; we assume the p-document given and investigate how to efficiently query it.

In contrast with existing work [1], [5], [9] that has proposed algorithms for tractable subcases and characterized the complexity of the problem, we consider in this work a very general form of p-documents (involving arbitrary correlations between nodes of the tree) together with a large class of queries (tree-pattern queries with joins, with results extensible to the even more general class of *locally monotone* [10] queries) and aim at a practical solution for querying p-documents. Computing the exact probability of a given answer to the query over a p-document is #P-hard [1], but, under data complexity, there are fully polynomial-time randomized approximation schemes [1]: we thus focus on efficiently approximating the probabilities of answers. Indeed, for many applications, one simply needs a good estimate of the probability value, i.e., an approximation *to a multiplicative factor* of the correct probability, *with high confidence*. Approximations *to an additive factor* are of lesser interest, since they make it hard to distinguish between, say, probabilities of 10^{-2} and of 10^{-5}.

Following ideas from [1], [10], we reduce the problem of approximating the probability of a given answer to a query Q over p-document $\mathscr{P}$ to approximating the probability of a propositional formula φ in disjunctive normal form (DNF). We first rewrite the initial (XPath) query Q into an XQuery query Q' that returns, for every match of Q over the deterministic document underlying $\mathscr{P}$, the conjunction of events conditioning this match (labeled probabilities related to the probabilistic nodes of the p-document). This rewritten query is then evaluated by a standard XQuery processor, allowing the use of all standard XML indexing techniques. This step can be done in time polynomial in the size of $\mathscr{P}$. The disjunction of all conjunctions of events associated to different probabilistic matches referring to the same item constitutes the propositional *lineage* φ of this item. Note that computing the probability of a propositional formula relates to computing the number of satisfying assignments of the formula, a problem known to be #P-complete [11].

Our system, ProApproX, has the characteristic of not relying on a single algorithm to evaluate the probability of a lineage formula but of deciding on the algorithm to be used based on a *cost model* of the various potential evaluation algorithms. In addition, the formula is compiled into an evaluation plan, different evaluation algorithms can be used on different subparts of the formula, and the overall cost of the whole plan is

estimated. As with regular query optimizers, the space of evaluation plans (for a user-specified approximation guarantee) is searched for one of optimal cost. ProApproX thus reveals its originality through the following major features:

1) The support of a broader range of XPath queries over a more general data model than current probabilistic XML systems (Sections II and III);
2) A cost model for a variety of probability evaluation algorithms, based on which the selection of the most efficient algorithm is performed (Section IV);
3) Simplification of the computation process via a decomposition of the probabilistic lineage into independent computational units (Section V);
4) The possibility of setting arbitrary error bound ε and confidence δ for the desired probabilistic approximation, with well-grounded propagation mechanisms of error and reliability between computational units (Section VI);
5) An exploration of the space of evaluation plans based on the proposed cost model (Section VII).

We start with a brief introduction to the probabilistic XML model, and to probabilistic lineage. After presenting all features of the algorithm, we present an extensive experimental evaluation of the proposed system in Section VIII, and go over the related work in Section IX before concluding.

II. Probabilistic XML

We recall here some basic notions about probabilistic XML as an uncertain data model [1], [4], [6].

Documents and p-Documents: We model XML documents as unranked, unordered, labeled trees. Not taking into account the order between sibling nodes in an XML document is a common but non-crucial assumption. The same modeling can be done for ordered trees, without much change to the theory.

A *probabilistic XML document* (or *p-document*) is similar to a document, with the difference that it has two types of nodes: ordinary and distributional. Distributional nodes are fictive nodes that specify how their children can be randomly selected. Given the criteria of dependency between elements, we distinguish two types of data representation models. In the *local dependency* model (named $\mathsf{PrXML}^{mux,ind}$ in [6]), choices made for different nodes are independent or locally dependent, i.e., a distributional node chooses one or more children independently from other choices made at other levels of the tree. In the *long-distance dependency* model [10] (PrXML^{cie} in [6]), the generation at a given distributional node might also depend on different conditions (i.e., probabilistic choices) related to other parts of the tree. This model, more general than the local dependency model, is based on one distributional node, *cie* (for *conjunction of independent events*): nodes of this type are associated with a conjunction of independent (possibly negated) random Boolean variables $e_1 \dots e_m$ called *events*; each event has a global and independent probability $\Pr(e_i)$ that e_i is true. Note that different *cie* nodes share common events, i.e., choices can be correlated.

The *semantics* of a p-document is a probability distribution over a set of possible documents, defined by the following process for the long-distance dependency case. First, randomly draw a truth assignment for each of the e_i's, following $\Pr(e_i)$. Then remove all distributional nodes whose condition is falsified by this assignment. Descendant of removed nodes are removed, and the remaining ordinary nodes are connected to their lowest ordinary ancestor, yielding a regular document. The probability of this document is that of all truth assignments that generate it.

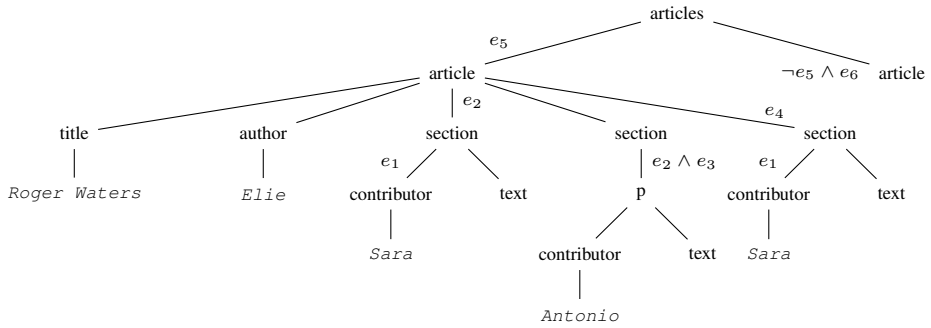
As noted in [6], local dependencies expressed by *ind* (independent) or *mux* (mutually exclusive) nodes can be tractably translated into *cie* nodes. We use these results to efficiently translate datasets in the local-dependency model into p-documents that rely exclusively on *cie* distributional nodes. This allows us to consider the long-distance dependency model only in the remaining of the paper.

As an example, Fig. 1 presents a fragment of a probabilistic XML document describing a given Wikipedia article as a merge of all its (uncertain) revisions, reproduced (with elaboration) from [8]. For ease of presentation, *cie* nodes are shown by simply annotating edges with conjunctions of literals. For example, the first "section" element in the tree appears under a *cie* node that retains it only if e_2 is true; the whole first "article" is kept if and only if e_5 is true; etc. Events e_i's appearing in the tree correspond to particular contributors, or probabilistic events that particular revisions are correct given that the authors are reliable [8], with the probability distribution of the $\Pr(e_i)$'s, on the right of the figure, typically inferred by a trust inference algorithm.

Querying Probabilistic XML: The query language we consider in this work is *tree-pattern queries with joins* [10]. A tree-pattern query with joins is given by a tree pattern (a tree whose edges are labeled with either *child* or *descendant* and whose nodes are labeled, possibly by a wildcard) together with a set of *value joins* that impose that two nodes of the tree pattern are matched to nodes with the same value. A *match* of a query to a (deterministic) document must map all nodes of the query to the document, respecting the following constraints: (i) the root is mapped to the root; (ii) child edges are mapped to edges of the tree; (iii) descendant edges are mapped to a sequence of child edges; (iv) non-wildcard labels are preserved; (v) nodes in a join condition must have the same label. As in [10], we view each match as the minimal subtree containing all nodes of a document that are mapped to a node of the query. Queries are given using the standard XPath syntax (note that some tree-pattern queries with joins cannot be expressed in XPath 1.0 but they can all be expressed in XPath 2.0).

Example 1: Given the p-document of Fig. 1, we can search for all contributors to a given article using the tree-pattern query Q_1: `//article[title='Roger Waters']//contributor`. Similarly, the tree-pattern query with join Q_2: `/articles/article[author=.//contributor]` looks for articles where the author appears among the contributors.

Though we focus on tree-pattern queries with joins for simplicity, a similar processing can be applied to a larger class of queries, namely *locally monotone* [10] queries.



Event	Prob.
e_1	0.2
e_2	0.6
e_3	0.5
e_4	0.7
e_5	0.9
e_6	0.4
...	

Fig. 1. Tree representation of a p-document with long-distance dependencies

III. Probabilistic XML Query Lineage

Recall the example query Q_1 that looks for the list of contributors to the revisions of the article entitled "Roger Waters". Looking at Fig. 1 we can see three matches to Q_1 on the deterministic document underlying the p-document, corresponding to the three different "contributor" nodes. Nevertheless, the system should return only two distinguished answers to the user: "`Sara`" and "`Antonio`". For each match, we can construct the conjunction of all probabilistic literals on the path from the root to one of the node matched by the query. These conjunctions are called the *probabilistic lineage* of each match. The first answer is about the contributor "`Sara`" that has two probabilistic instances over the tree, and a corresponding probabilistic lineage of:

$$\varphi_{Sara} = c_1 \vee c_3 = (e_5 \wedge e_2 \wedge e_1) \vee (e_5 \wedge e_4 \wedge e_1).$$

The remaining item, "`Antonio`", appears once and has thus a DNF lineage composed only of the clause c_2:

$$\varphi_{Antonio} = c_2 = (e_5 \wedge e_2 \wedge e_3).$$

Because we only have conjunctions of literals on distributional nodes in PrXML^{cie}, probabilistic lineages are always in disjunctive normal form (DNF). In the following, we will often note this DNF $\varphi = \bigvee_{i=1}^{m} c_i$ and will call each c_i a *clause* of the DNF. In ProApproX, probabilistic events for a given node are stored in an attribute node named "prob" of a regular XML document, stored and managed by an ordinary native XML DBMS. To retrieve items' lineages, we need to transform our XPath query Q into an XQuery query Q' that returns the set of items that match the user query Q, together with their corresponding DNF lineage (i.e., for each instance of an item, returns the concatenation of all "prob" attributes of each match). For instance, for Q_1, we obtain Q'_1:

```
for $val in distinct-values
(//article[title='Roger Waters']//contributor)
order by $val
  return <match>{$val}{
   for $a in //article
   for $b in $a/title/text()[.='Roger Waters']
   for $c in $a//contributor
   let $leaves:=($b,$c)
   let $atts:=(for $l in $leaves
     where $c=$val (: The grouping function :)
     return $l/ancestor-or-self::*/@prob)
   return <clause>{distinct-values
     (for $att in $atts return string($att))
     }</clause>}
  </match>
```

A generic translator from tree-pattern queries with joins encoded in XPath into XQuery lineage queries is implemented as part of ProApproX.

We can proceed similarly for *Boolean* queries, queries that test for the match of a tree pattern. If Q_1 is seen as a tree-pattern query, its probabilistic lineage over the document of Fig. 1 is obviously $\varphi_1 = c_1 \vee c_2 \vee c_3$, and a variation on Q'_1 above produces this lineage.

We have thus separated the problem of querying probabilistic XML into two independent problems: (i) evaluating an XQuery query over a regular XML document; (ii) computing the probability of a formula in DNF in order to obtain the probability of a given answer. The former problem can be solved using any XQuery processor; we use the native XML DBMS BaseX[1] after comparing its performance on our datasets to other XQuery engines (in particular, to another native XML DBMS, eXist[2], and to an in-memory XQuery processor, SAXON[3]). The latter problem is the focus of the remaining of this paper.

Computing the exact probability of a formula in DNF is a #P-hard problem in general, even when all events have the same probability [11], and it is easy to see that any DNF formula can be generated as the lineage of even a trivial XPath query such as `/A`, see [5]. However, we are going to detect some easy cases and exploit polynomial-time randomized approximation algorithms [1] to build an efficient query processor over probabilistic XML.

IV. Algorithms and Cost Models

In this section we present a collection of algorithms for computing or approximating the probability of a DNF formula φ. Based on the computational complexity of each of these algorithms, we elaborate a cost model that estimates the runtime (in ms) of each algorithm alg as a function cost_{alg} of different features of φ, of approximation parameters defined further, and of a cost constant C_{alg} representing the time needed for the elementary, inner, parts of each algorithm.

[1] http://basex.org/
[2] http://exist-db.org/
[3] http://saxon.sourceforge.net/

TABLE I
NOTATION

φ	DNF formula $\varphi = \bigvee_{i=1}^{m} c_i$
m	number of clauses in φ
N	number of event variables used in φ
c_i	a clause of φ
k_i	size of clause c_i
L	total size of φ, $L = \sum_{i=1}^{m} k_i$
t	number of trials (samples)
ε	approximation error
$1-\delta$	probabilistic approximation guarantee
ℓ	lower bound on φ's probability
cost_{alg}	cost of an algorithm alg
C_{alg}	cost constant for algorithm alg

TABLE II
COST MODELS AND COST CONSTANTS

Algorithm *alg*	cost_{alg}	C_{alg} (ms)
naïve	$C_{\text{naïve}} \times 2^N \times L$	$4 \cdot 10^{-5}$
sieve	$C_{\text{sieve}} \times 2^m \times \frac{L}{m}$	$5 \cdot 10^{-5}$
AddMC	$C_{\text{AddMC}} \times \ln \frac{2}{\delta} \times \frac{L}{\varepsilon^2}$	$4 \cdot 10^{-5}$
MulMC	$C_{\text{AddMC}} \times \ln \frac{2}{\delta} \times \frac{L}{\ell^2 \varepsilon^2}$	$4 \cdot 10^{-5}$
coverage	$C_{\text{coverage}} \times \ln \frac{2}{\delta} \times \frac{(1+\varepsilon)\times L}{\varepsilon^2}$	10^{-3}

A value for C_{alg} is measured experimentally by varying the size of synthetic formulas (see Section VIII for precisions on generation) and recording the processing time of the actual implementation of the algorithm on a given machine. We only consider sequential processing here, see the conclusion for discussions about parallelization. We give in Table I a summary of the notation used and in Table II a summary of the cost models of all algorithms.

A. Exact Computation

We start with exact computation algorithms. Since the problem is #P-hard, only exponential-time algorithms are known, but they are typically very fast on DNFs of small size. We present two such algorithms, that are folklore, and are well-suited, respectively, to the case when there are few variables, or few clauses, in φ.

Possible worlds (Naïve algorithm): This algorithm corresponds to the naïve, exponential-time, iteration over all possible 2^N truth value assignments to the N variables used in φ, i.e., over all possible worlds. It simply consists in summing up the probabilities of the satisfying assignments. In the example of the previous sections, the formula $\varphi_1 = (e_5 \wedge e_2 \wedge e_1) \vee (e_5 \wedge e_2 \wedge e_3) \vee (e_5 \wedge e_4 \wedge e_1)$ has 5 different variables, and we can enumerate all 32 possible worlds; 8 of them make φ_1 true, and one can check that their total probability is 0.3744.

The computational complexity of the naïve algorithm is obviously $O(2^N \times L)$: we enumerate all possible worlds, and for each one we evaluate the truth value of the formula in linear time. We therefore set the following cost:

$$\text{cost}_{\text{naïve}} = C_{\text{naïve}} \times 2^N \times L$$

where $C_{\text{naïve}}$ is determined experimentally as previously explained and is equal to $4 \cdot 10^{-5}\,\text{ms}$ (see Table II for all cost constants). This means, for instance, that the expected runtime of the naïve algorithm for a formula with 10 variables and of length 500 is $C_{\text{naïve}} \times 2^{10} \times 500 \approx 20\,\text{ms}$ (for the machine for which the cost constants were computed), which is reasonably low and shows that, at least for some DNF formulas, the naïve algorithm can be appropriate.

Inclusion–exclusion (sieve): It is also possible to apply the *inclusion–exclusion* or *sieve* principle to compute the probability of φ. The *sieve* decomposition of the DNF φ is:

$$\Pr\left(\bigvee_{i=1}^{n} c_i\right) = \sum_{k=1}^{n} (-1)^{k-1} \sum_{\substack{I \subset \{1,\ldots,n\} \\ |I|=k}} \Pr(c_I), \text{ where } c_I := \bigwedge_{i \in I} c_i.$$

On our example, $\Pr(\varphi_1) = \Pr(c_1) + \Pr(c_2) + \Pr(c_3) - \Pr(c_1 \wedge c_2) - \Pr(c_2 \wedge c_3) - \Pr(c_1 \wedge c_3) + \Pr(c_1 \wedge c_2 \wedge c_3) = 0.108 + 0.27 + 0.126 - 0.054 - 0.0378 - 0.0756 + 0.0378 = 0.3744$. The computational complexity is $O(2^m \times \frac{L}{m})$: the probability of the conjunction of each set of clauses, whose typical size is of the order of $\frac{L}{m}$, can be computed in linear time. We set:

$$\text{cost}_{\text{sieve}} = C_{\text{sieve}} \times 2^m \times \frac{L}{m},$$

which becomes competitive with respect to the naïve algorithm when clauses are few but long.

Note that, as described here, the sieve method is numerically instable. Implementing the sieve formula presented earlier in floating-point arithmetic results in a low accuracy because of rounding errors in sequences of additions and deletions. The sieve method can be improved towards better numerical stability using an algorithm proposed by Heidtmann [12].

B. Approximation Algorithms

We present in this section two randomized approximation algorithms, giving respectively *additive* and *multiplicative* guarantees. For fixed ε, δ, we say $A(\varphi)$ is an additive ε-approximation of $\Pr(\varphi)$ with probabilistic guarantee $1-\delta$ if, with probability at least $1-\delta$, the following holds:

$$\Pr(\varphi) - \varepsilon \leqslant A(\varphi) \leqslant \Pr(\varphi) + \varepsilon.$$

Similarly, for fixed ε and δ, $A(\varphi)$ is a multiplicative ε-approximation of $\Pr(\varphi)$ with probabilistic guarantee $1-\delta$ if, with probability at least $1-\delta$:

$$(1-\varepsilon) \times \Pr(\varphi) \leqslant A(\varphi) \leqslant (1+\varepsilon) \times \Pr(\varphi).$$

In Section VI, we explain how to turn additive guarantees into multiplicative ones.

Additive Monte-Carlo: The simplest way to implement an additive approximation is by sampling, Monte-Carlo–style, the space of all possible assignments following the distribution of the literals, and test whether the picked assignment satisfies at least one clause. If we conduct t trials, an (unbiased) estimator $A(\varphi)$ for the probability of φ will be the proportion of the

trials that led to a satisfaction of φ. Following Hoeffding's bound [13], we have:

$$\Pr\left(|A(\varphi) - \Pr(\varphi)| > \varepsilon\right) \leqslant 2\mathrm{e}^{-2t\varepsilon^2}.$$

Therefore, the number of trials t needed to perform an (ε, δ)-additive approximation is:

$$t = \left\lceil \frac{\ln 2 - \ln \delta}{2\varepsilon^2} \right\rceil.$$

For each sample, the algorithm linearly scans the formula, which gives an overall complexity of $O(t \times L)$ and a cost model:

$$\mathrm{cost}_{\text{AddMC}} = C_{\text{AddMC}} \times \ln \frac{2}{\delta} \times \frac{L}{\varepsilon^2}.$$

Self-Adjusting Coverage Algorithm: This algorithm was introduced by Karp, Luby, and Madras [14] and is a fully-polynomial randomized approximation scheme (FPRAS), which means it produces a multiplicative estimate of the probability. We base our cost model on the following result:

SELF-ADJUSTING COVERAGE THEOREM I [14]. *When* $\varepsilon < 1$ *and* $t = (8 \times (1+\varepsilon) \times m \ln(2/\delta))/\varepsilon^2$*, the self-adjusting coverage algorithm yields an* (ε, δ)*-approximation.*

Since each trial requires evaluating one clause of φ, of average size $O(\frac{L}{m})$, we have:

$$\mathrm{cost}_{\text{coverage}} = C_{\text{coverage}} \times \frac{L}{m} \times \frac{(1+\varepsilon)m}{\varepsilon^2} \times \ln \frac{2}{\delta}.$$

As can be seen on Table II, the cost constant computed for this self-adjusting coverage algorithm is much higher than the cost constants for other algorithms, reflecting the fact that though it has an excellent algorithmic complexity (indeed, practically the same as Monte-Carlo sampling, which only yields an additive approximation), this can still be a costly algorithm to use in practice.

As we shall see, each of these four algorithms is most efficient on some of the possible φ's. However, we do not apply them on the whole formula, but start by decomposing φ into an evaluation tree built out of simpler formulas.

V. BUILDING THE EVALUATION TREE

We explain in this section how ProApproX decomposes a DNF formula φ into an evaluation tree formed of simpler subformulas. We assume here that no contradiction appears in φ (a clause that contains both x and $\neg x$ is removed from φ in a preprocessing step). To illustrate, we use the following running example:

Example 2: Consider the formula $\varphi_2 = (e_1 \wedge e_2) \vee (e_1 \wedge e_3) \vee (\neg e_4 \wedge e_5) \vee (e_4 \wedge e_6) \vee (e_4 \wedge e_7)$. To compute $\Pr(\varphi_2)$, we rewrite φ_2 as the composition of subformulas whose probabilities are simpler to compute.

Our evaluation tree is based on three operations (independence detection, factorization, inconsistency detection) that are repeatedly applied on the original formula. All three operations yield trees whose leaves are formula in DNF and whose internal nodes are Boolean operations (independent disjunction $\textcircled{\vee}$, independent conjunction $\textcircled{\wedge}$, and mutually exclusive disjunction $\bigoplus$) that support efficient probability computations.

Independence: We say that two clauses are *independent* if they do not share any variables. Our first operation attempts to write the DNF formula φ as an *independent disjunction* $\varphi' \textcircled{\vee} \varphi''$ of two DNF formulas that form a partition of φ such that every clause of φ' is independent from every clause of φ''. In our example, we can write φ_2 as $((e_1 \wedge e_2) \vee (e_1 \wedge e_3)) \textcircled{\vee} ((\neg e_4 \wedge e_5) \vee (e_4 \wedge e_6) \vee (e_4 \wedge e_7))$.

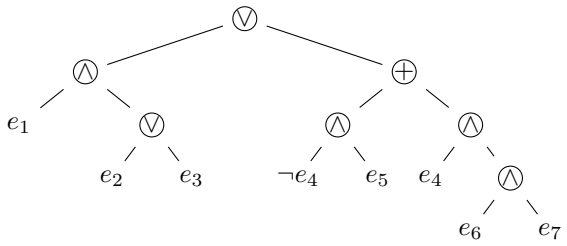
Factorization: Our second operation consists in *factoring* the intersection c of all clauses c_i of formula φ out of the clauses, obtaining a rewriting of φ as $\varphi' = c \textcircled{\wedge} (c'_1 \vee \cdots \vee c'_m)$ where $\textcircled{\wedge}$ denotes independent conjunction: c is independent of every c_i. Note that both operands of $\textcircled{\wedge}$ are in DNF: the left-hand side is a simple conjunctive clause, and the right-hand side is a DNF formula. If we apply this factorization to the subformulas appearing in our previous rewriting of φ_2, we obtain $(e_1 \textcircled{\wedge} (e_2 \vee e_3)) \textcircled{\vee} ((\neg e_4 \wedge e_5) \vee (e_4 \wedge e_6) \vee (e_4 \wedge e_7))$.

Inconsistency: Our final operation looks for a partition of a DNF formula into two inconsistent subformulas, i.e., a rewriting of φ into $\varphi' \bigoplus \varphi''$ where φ' and φ'' are partitions of the clauses of φ, and there is a variable x such that all clauses of φ' have a literal $\neg x$ and all clauses of φ'' have a literal x. Our example formula becomes $(e_1 \textcircled{\wedge} (e_2 \vee e_3)) \textcircled{\vee} ((\neg e_4 \wedge e_5) \bigoplus ((e_4 \wedge e_6) \vee (e_4 \wedge e_7)))$.

Evaluation tree: ProApproX repeatedly applies these three operations until no further rewritings are possible. We obtain thus for φ_2:

$$(e_1 \textcircled{\wedge} (e_2 \textcircled{\vee} e_3)) \textcircled{\vee} ((\neg e_4 \textcircled{\wedge} e_5) \bigoplus (e_4 \textcircled{\wedge} (e_6 \textcircled{\vee} e_7))).$$

This defines an evaluation tree for φ_2 in a straightforward manner; inner nodes are the Boolean operations $\textcircled{\vee}$, $\textcircled{\wedge}$, $\bigoplus$ (assumed binary for simplicity), and leaves are DNF formulas that cannot be simplified any further (in the simple case of φ_2, these are just trivial formulas formed of a single literal):



Note that the process is nondeterministic, since it depends on the order of application of operations. We choose arbitrarily one such order (independence, factorization, inconsistency, and then the whole sequence again, until reaching a fixpoint, which necessarily happens since all operators produce smaller formulas than the original one). We do not make any claim of optimality of the simplification obtained but observe that, at least, leaf formulas obtained are independent of the order (though the structure of the tree may differ):

Proposition 1: The set of formulas contained in leaf nodes of the evaluation tree obtained by repeatedly applying the three independence, factorization, inconsistency operator in whatever order until reaching a fixpoint is independent of the order.
This can be proved by an induction on the size of the formula.

Our evaluation trees are essentially special cases of *d-trees*, the decision diagrams used in the SPROUT [3] probabilistic query engine, where we disallow Shannon expansion – as a consequence, our evaluation trees always have a size bounded by that of the original formula, in contrast with d-trees.

Evaluation trees can be used to compute the probability of the main formula in terms of the probabilities of the formulas on the leaves, thanks to the following observations:

$$\Pr(\psi_1 \otimes \psi_2) = 1 - (1 - \Pr(\psi_1)) \times (1 - \Pr(\psi_2))$$
$$\Pr(\psi_1 \oslash \psi_2) = \Pr(\psi_1) \times \Pr(\psi_2)$$
$$\Pr(\psi_1 \oplus \psi_2) = \Pr(\psi_1) + \Pr(\psi_2)$$

Towards an evaluation plan: In general, the evaluation tree of a formula still contains as leaves DNF formulas that are hard to compute. To compute the probability of the global formula, we turn these evaluation trees into evaluation plans by assigning to every leaf of the tree one algorithm (either exact or approximate) that will be used to compute the probability of this leaf. This assignment will obviously use the cost model for the different algorithms, but we need to be careful about the following aspects of the problem: (i) We need to propagate approximation parameters down the tree in a principled manner, so that we keep the approximation guarantees on the global probability (see next section). (ii) It is sometimes worth not going down to the level of leaves of the approximation tree to evaluate the probabilities, but to do it at a higher level; in other words, we might want to assign evaluation algorithms to internal nodes of the tree rather than to leaves (see Section VII).

VI. Propagation of approximation parameters

Our overall objective is to obtain a multiplicative approximation of the probability of a formula φ in DNF, given an approximation tolerance ε and probabilistic approximation guarantee $1 - \delta$. We have explained in the previous section that we will achieve this by decomposing φ into its evaluation tree and assigning different algorithms, some exact, and some approximate, to different parts of the tree. These assignments must be decided on so that the produced precisions at each node yield an overall approximation that does not exceed the tolerance set for the approximation of the DNF formula as a whole. In this section, we establish the different conditions for correct error allocation for children of $\otimes$, $\oslash$, or $\oplus$ nodes, in the case when the probability of both nodes is approximated to a multiplicative factor (conditions for additive approximations can be found in [15]). We also explain how the approximation guarantee $1 - \delta$ is propagated, and how to turn an additive approximation into a multiplicative one.

Propagation of approximation error ε:

Proposition 2: Let $\varphi = \psi_1 \otimes \psi_2$, and assume $\tilde{p_1}$ and $\tilde{p_2}$ are multiplicative approximations of $\Pr(\psi_1)$ and $\Pr(\psi_2)$, to a factor of ε_1 and ε_2, respectively. Then $1 - (1 - \tilde{p_1})(1 - \tilde{p_2})$ is a multiplicative approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon = \varepsilon_1 + \varepsilon_2$.

Proof: Let us denote p the probability of φ and similarly p_1 and p_2 the probabilities of ψ_1 and ψ_2; $\tilde{p_1}$ and $\tilde{p_2}$ are multiplicative approximations of p_1 and p_2, and

$$\begin{cases} (1 - \varepsilon_1)p_1 \leqslant \tilde{p_1} \leqslant (1 + \varepsilon_1)p_1 \\ (1 - \varepsilon_2)p_2 \leqslant \tilde{p_2} \leqslant (1 + \varepsilon_2)p_2 \end{cases}$$

Then:

$$\begin{cases} 1 - (1 + \varepsilon_1)p_1 \leqslant 1 - \tilde{p_1} \leqslant 1 - (1 - \varepsilon_1)p_1 \\ 1 - (1 + \varepsilon_2)p_2 \leqslant 1 - \tilde{p_2} \leqslant 1 - (1 - \varepsilon_2)p_2 \end{cases}$$

And thus:

$$1 - [(1 - (1 - \varepsilon_1)p_1)(1 - (1 - \varepsilon_2)p_2)] \leqslant 1 - (1 - \tilde{p_1})(1 - \tilde{p_2})$$
$$1 - (1 - \tilde{p_1})(1 - \tilde{p_2}) \leqslant 1 - [(1 - (1 + \varepsilon_1)p_1)(1 - (1 + \varepsilon_2)p_2)]$$

Let α be $1 - [(1 - (1 - \varepsilon_1)p_1)(1 - (1 - \varepsilon_2)p_2]$. We have:

$$\begin{aligned} \alpha &= 1 - [(1 - p_1 + \varepsilon_1 p_1)(1 - p_2 + \varepsilon_2 p_2)] \\ &= \underbrace{1 - (1 - p_1)(1 - p_2)}_{=\mathsf{p}} \\ &\qquad - [(1 - p_1)\varepsilon_2 p_2 + (1 - p_2)\varepsilon_1 p_1 + \varepsilon_1 \varepsilon_2 p_1 p_2]. \end{aligned}$$

Let $\lambda = [(1 - p_1)\varepsilon_2 p_2 + (1 - p_2)\varepsilon_1 p_1 + \varepsilon_1 \varepsilon_2 p_1 p_2]$. $\frac{\lambda}{p} = \frac{(1-p_1)\varepsilon_2 p_2 + (1-p_2)\varepsilon_1 p_1 + \varepsilon_1 \varepsilon_2 p_1 p_2}{p_1 + p_2 - p_1 p_2}$. Let $\Pi = \max(p_1, p_2)$; we have $p \geqslant \Pi$. We write λ as $\varepsilon_1 p_1[1 + p_2(\varepsilon_2 - 1)] + \varepsilon_2 p_2(1 - p_1)$; clearly, $\lambda \leqslant \varepsilon_1 \Pi + \varepsilon_2 \Pi$. Then $\frac{\lambda}{p} \leqslant (\varepsilon_1 + \varepsilon_2)\frac{p}{p} \leqslant \varepsilon_1 + \varepsilon_2 = \varepsilon$ and $\lambda \leqslant \varepsilon p$.

We proceed similarly for the upper bound. ∎

In practice, when we propagate an ε factor down the tree, we look for different combinations of ε_1 and ε_2 that satisfy a given propagation condition at a parent node, evaluate the cost of each combination, and choose the best found (this is one example of the general exploration of evaluation plans described in the next section). Note that if $\varepsilon_1 = 0$ (that is, if we compute the exact value of $\Pr(\psi_1)$), we can set $\varepsilon_2 = \varepsilon$.

Proposition 3: Let $\varphi = \psi_1 \oplus \psi_2$, and assume $\tilde{p_1}$ and $\tilde{p_2}$ are multiplicative approximations of $\Pr(\psi_1)$ and $\Pr(\psi_2)$, to a factor of ε_1 and ε_2, respectively. Then $\tilde{p_1} + \tilde{p_2}$ is a multiplicative approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon = \max(\varepsilon_1, \varepsilon_2)$.

Proof: We use the same notation as before.

$$\begin{cases} p_1 - \varepsilon_1 p_1 \leqslant \tilde{p_1} \leqslant p_1 + \varepsilon_1 p_1 \\ p_2 - \varepsilon_2 p_2 \leqslant \tilde{p_2} \leqslant p_2 + \varepsilon_2 p_2 \end{cases}$$

and then:

$$p_1 + p_2 - \varepsilon_1 p_1 - \varepsilon_2 p_2 \leqslant \tilde{p_1} + \tilde{p_2} \leqslant p_1 + p_2 + (\varepsilon_1 p_1 + \varepsilon_2 p_2)$$

Note that $(\varepsilon_1 p_1 + \varepsilon_2 p_2) \leqslant \varepsilon(p_1 + p_2) = \varepsilon p$ which gives the required lower bound; the upper bound is similar. ∎

In particular, if $\varepsilon_1 = 0$ we can set again $\varepsilon_2 = \varepsilon$.

For the $\oslash$ operator, observe that it never appears between two arbitrary DNF formulas, but between a formula in DNF and a conjunction; the probability of the latter can be evaluated exactly in a tractable manner. We thus just state the propagation condition in this case:

Proposition 4: Let $\varphi = \psi_1 \oslash \psi_2$, and assume $\tilde{p_1}$ is a multiplicative approximation of $\Pr(\psi_1)$ to a factor of ε_1. Then $\tilde{p_1} \times \Pr(\psi_2)$ is a multiplicative approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon = \varepsilon_1$.

Propagation of approximation guarantee $1-\delta$*:* The propagation of the guarantee $1-\delta$ is quite straightforward. We assume approximations of different subformulas are going to be carried out in an independent manner, using different samples. Then, if:

$$\begin{cases} \Pr(|p_1 - \tilde{p_1}| \leqslant \varepsilon_1 p_1) \geqslant 1-\delta_1 \\ \Pr(|p_2 - \tilde{p_2}| \leqslant \varepsilon_2 p_2) \geqslant 1-\delta_2 \end{cases}$$

we have:

$$\Pr((|p_1-\tilde{p_1}| \leqslant \varepsilon_1 p_1) \wedge (|p_2-\tilde{p_2}| \leqslant \varepsilon_2 p_2)) \geqslant (1-\delta_1)(1-\delta_2).$$

This gives the propagation rule: $1-\delta = (1-\delta_1)(1-\delta_2)$.

In ProApproX, in any case where we use approximations for both operands of one of the internal nodes of the evaluation tree, we simply set $\delta_1 = \delta_2 = 1-\sqrt{1-\delta}$.

Multiplicative guarantee from additive approximations: Resorting to an execution plan using additive algorithms with the ultimate goal of producing a multiplicative tolerance, might sometimes be more efficient then running a multiplicative approximation. In that case, we would like the result to be within a multiplicative error interval $[p-p\varepsilon, p+p\varepsilon]$ of the probability p, for a given ε. Thus, we need to set an input error ε_{add} for the additive algorithm, so that: $\varepsilon_{\text{add}} = \varepsilon p$. It is not possible to exactly determine this value since p is the quantity that we are actually looking for. What we propose is to use a *lower bound* ℓ for p and set ε_{add} to $\ell\varepsilon$. This guarantees us that ε_{add} is small enough to obtain a multiplicative error of ε. One such simple lower bound, used in ProApproX, is the maximum probability of a clause of the DNF, that can be easily determined by a linear scan of the DNF.

Using this trick, we can see Monte-Carlo sampling as a multiplicative approximation algorithm, with a new cost model of $\text{cost}_{\text{MulMC}} = C_{\text{AddMC}} \times \ln\frac{2}{\delta} \times \frac{L}{\ell^2\varepsilon^2}$.

VII. Evaluation Plans

We now have all necessary components for defining *evaluation plans* of a DNF formula φ. An (ε,δ)-evaluation plan for φ is a subtree of the evaluation tree of φ presented in Section V, with the same root, where every leaf is associated with an algorithm among the four described in Section IV, and, when this algorithm is an approximation algorithm, with approximation parameters ε', δ'. We further require that these approximation parameters guarantee a multiplicative (ε,δ)-approximation of φ, following the constraints of Section VI. The cost of a leaf of an evaluation plan is the cost of applying the given algorithm on the formula at that leaf. The cost of an evaluation plan is defined as the sum of the costs of all leaves (we thus ignore the cost of combining the probabilities at each internal node, which amounts to evaluating a constant number of arithmetic operations, see Section V). There are many (indeed, infinitely many in general) evaluation plans for a given formula φ, corresponding to different subtrees of the evaluation tree, and to different assignments of approximation parameters. Our goal is to find one of minimum cost.

When propagating an error bound ε from one node to its children, many possible different values for ε_1 and ε_2 can be used (e.g., under the condition $\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2 \leqslant \varepsilon$ for two approximated nodes linked by and independent disjunction, different values for ε_1 and ε_2 satisfy the condition). We can find the best assignment locally for the direct descendant of a node by finding the global minimum of the (ε_1,ε_2) function. However, this does not necessarily guarantee producing the most optimal propagation for the whole tree. Indeed, if we decide also to vary the allocation of these parameters, the number of possible plans is unbounded. Inspired by query optimization techniques, we can explore the space of evaluation plans for such a case by sampling a large number of them, evaluating the cost of each, and choosing the best one found, which is thus not necessarily optimal, but can be good enough in practice.

After experimenting with such a sampling technique (see [15] for details), we decided instead to use in ProApproX a more deterministic approach: we first decide on an adequate allocation of the approximation parameters, and then look for the best evaluation plan with respect to these. Our exploration of the search space is based on the following principles. We start with the evaluation tree, and global ε, δ:

1) First visit of the tree (top-down): We recursively assign ε and δ values down the tree, following the logical operator at the parent node, and propagation rules from Section VI; we always set $\varepsilon_1 = \varepsilon_2$ and $\delta_1 = \delta_2$;
2) Second visit of the tree (bottom-up):
   a) We assign the best evaluation algorithm for a given node, following local parameters and based on the cost model;
   b) If a given node is assigned with an exact algorithm, we re-assign unused ε, δ to nodes already assigned with an approximation algorithm, or to remaining nodes of the tree;
   c) We restart (a,b) for the next node (including inner nodes);
3) Third visit of the tree (bottom-up): We then decide whether we want to chose to apply the algorithm at a given node (and forget about the algorithms assigned to its descendants) or at one of its ancestors.

Example 3: Fig. 2 illustrates a mock-up example of a best evaluation plan, or best-tree, where computation nodes are marked with flags (framed nodes in the example compilation tree). Note that the algorithm for finding the best-tree in the deterministic case amounts to comparing the cost of a node with that of its children (a child node is never more expensive than its parent; however, the sum of children costs can exceed a parent-node cost).
As we shall see, proceeding this way leads to better results than always choosing, say, the full evaluation tree.

VIII. Experiments

In this section we evaluate our approach experimentally. We address the following questions: (1) How does our new query evaluation method compare to the state of the art in probabilistic XML database query processing? (2) How accurate are the results? (3) How effective is the tree compilation of the DNF

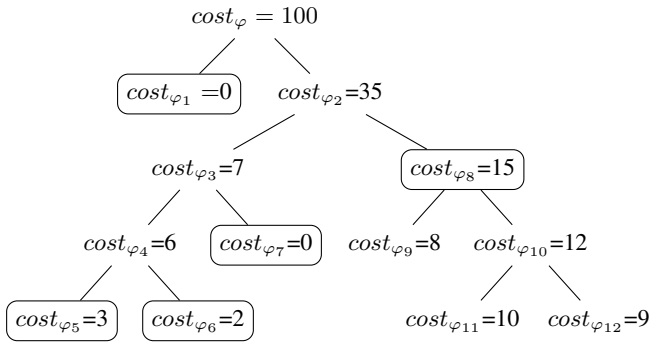


Fig. 2. Example best evaluation plan obtained by the deterministic procedure

formula over simply performing an exact or an approximate computation of the whole formula?

Experimental Setup: We implemented our system in Java. XQuery lineage queries, translated from the original XPath queries, are evaluated over the XML dataset through the XML::DB API for BaseX, the native XML DBMS. Using this API gives us the flexibility of plugging in an arbitrary XML database engine. All indexing features of BaseX (in particular, path summaries and text node index) were turned on. The resulting DNF formulas are then compiled into trees, potential evaluation strategies are explored and their cost assessed, and the chosen strategy is then run to evaluate the probability of the query. Our experiments were run on a desktop PC with an Intel Xeon CPU at 2.40 GHz running 64 bit Linux, with the data accessed by BaseX located on a standard magnetic disk.

Data and Queries: Our evaluation was carried out on three different datasets, two of them previously used in the probabilistic XML literature, the third one a synthetic dataset.

1) We used the *MondialDB* probabilistic XML document with local dependencies and 15 tree-pattern queries that served to evaluate EvalDP [1], [5]. EvalDP is a linear-time bottom-up algorithm for evaluating tree-pattern queries without joins on local-dependency p-documents. The queries have between two to six nodes depending on their complexity and selectivity. The p-document, also provided by the authors of [1], has been created by adding random probabilistic choices in a regular XML document. We added two extra queries with a join (`/mondial//country[.//name=@name]` and `/mondial//country[.//@capital=.//@id]`) to showcase the fact that ProApproX is able to deal with join and no-join queries indifferently.
2) We also obtained the *Movies* dataset and queries, used by Hollander and Van Keulen in [2], from the authors of that work. [2] explores the feasibility of storing and querying probabilistic XML documents in a probabilistic relational database. The authors considered real-life uncertain data obtained from a probabilistic data integration application [7], that produces a local-dependency p-document. The application integrates movie data from a TV guide[4] with that of the Internet Movie Database[5]. As in [2], the queries were ran over five p-documents of different sizes.
3) Finally, we tested the performance of our DNF probability evaluator over *synthetic* data: random DNF formulas with a number of literals L ranging from 1 to 700,000. Specifically, for a given target size number of literals, we iteratively randomly drew clauses in the following manner, until the target size was reached: first, draw a random clause size uniformly at random between 1 and $[L/3, L/10,$ and $L/100]$; second, uniformly draw positive or negative literals from a fixed set of variables (of size $L/2$) avoiding contradicting literals.

Methodology: Each query is translated into an XQuery lineage query that is evaluated on BaseX to capture the DNF lineage. The time of this operation is what we call *XQuery time*. To compare our approach to baselines, we first run the four main computation algorithms over the entire DNF lineage: sieve and naïve algorithms, Monte-Carlo sampling, and the self-adjusting coverage algorithm. The computation is stopped at one minute if not finished before. Afterwards, we fully compile the DNF in a tree structure. We then use our cost model to determine what the best strategy to evaluate the probability of each leaf. We call this the *full-tree* strategy. Finally, we used the exploration strategy detailed in Section VII to find, if possible, a tree whose cost is lower than both the cost of evaluating the whole DNF with any of the algorithms and the cost of the full tree strategy. This is the *best-tree* strategy, for which we can distinguish between the *Compilation time* spent in decomposing the DNF using the three presented operators in Section V, the *Exploration time* (to decide of the best tree and of the best allocation of error bounds and evaluation algorithms) and the *Evaluation time* itself, when the actual evaluation algorithms are run and the final probability is obtained. The running time of the best-tree strategy, ProApprox's default behavior, is thus obtained by summing *Compilation time*, *XQuery time*, *Exploration time*, and *Evaluation time*. In all experiments, we set the parameters for a multiplicative approximation with $\varepsilon = 0.1$ and a reliability factor $\delta = 95\%$.

Baselines: In addition to the simple algorithms already mentioned, we compare to three different state-of-the-art systems for probabilistic database querying, whenever possible:

EvalDP [1], a linear-time query evaluator for tree-pattern queries (without joins) over local-dependency p-documents. Because of these two restrictions, this algorithm is only applicable for the part of the MondialDB dataset that does not contain joins. We use the implementation kindly provided by the authors of [1].

Trio [16] is a probabilistic relational DBMS that was used in [2] to encode the data and queries of the Movies dataset, using an ad-hoc encoding for this particular dataset. We did not reimplement this encoding and just take the experimental figures from [2] (they are thus not completely comparable, but the order of magnitude remains significant).

[4]http://www.tvguide.com/

[5]http://www.imdb.com/

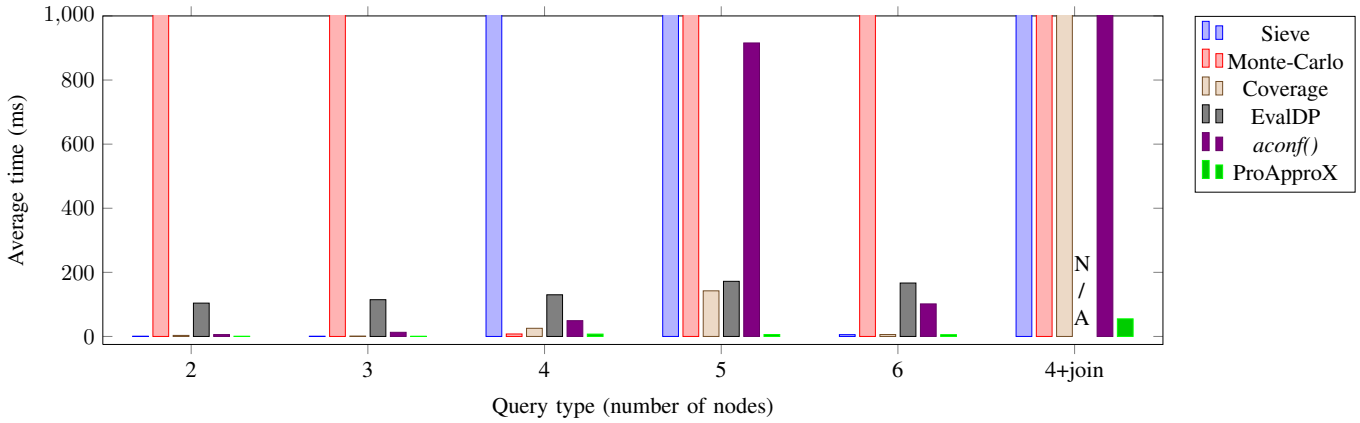


Fig. 3. Running time of the different algorithms on the MondialDB dataset [1], [5]

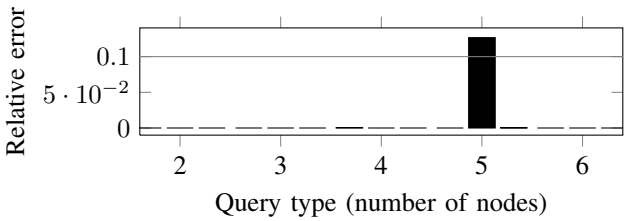


Fig. 4. Relative error on the probabilities computed by ProApproX on MondialDB over each non-join query ($\varepsilon = 0.1$, $\delta = 95\%$)

TABLE III
PROPORTION OF THE TIME SPENT ON EACH PART OF THE QUERY EVALUATION ON THE MONDIALDB DATASET (COMP: COMPILATION TIME, EXP: EXPLORATION TIME, EVAL: EVALUATION TIME)

Query type	Avg DNF size	XQuery	Comp	Exp+Eval
2	6	96.82%	2.47%	0.71%
3	5	98.19%	1.38%	0.42%
4	43	97.41%	2.01%	0.58%
5	252	64.47%	33.76%	1.78%
6	6	99.69%	0.29%	0.03%
4+join	3656	61.49%	36.12%	2.39%

SPROUT [3] is the query engine of the PostgreSQL-based MayBMS probabilistic DBMS [17], and the state-of-the-art in probabilistic database querying. To use it for our purpose, we first retrieved the DNF lineage of the queries over the p-document, and then encoded this DNF lineage as MayBMS relations (each clause is a row of the relation, whose arity is proportional to the maximum number of literals in a clause). This required some modifications to the latest available version of SPROUT[6] since it did not support out-of-the-box relations of arbitrary arity; we also run into some PostgreSQL configuration issues in the process. To get fair timings that do not include any disk-management PostgreSQL-specific time, we also inserted snippets of code to record the time needed to run the probability computation (compilation and evaluation of the lineage) and only that. We use the two-argument *aconf()* function that approximates the probability of a query in MayBMS (using an optimal algorithm based on Monte-Carlo sampling [3]), with the same ε, δ parameters as for ProApproX[7]. Since this baseline also requires first extracting the DNF lineage, we add to *aconf()* timings the XQuery time as well, as we do for ProApproX. The number of literals per clause in the synthetic dataset proved too large for MayBMS, so we only report timings for MondialDB and Movies.

MondialDB: We show experimental results of our system, and baselines on all MondialDB queries, grouped by type, in Fig. 3. The *best tree* strategy of ProApproX has highly competitive execution time, being in all cases under 100ms and the fastest probability computation method, sometimes tied with one of the baselines. The best baseline algorithm depends on the query: when the lineage is simple (types 2, 3, or 6) exact naïve or sieve computation methods are extremely fast, and this is the choice that our optimizer selects. On query type 4, Monte-Carlo approximations are fast (usually the sign of a high lower bound on the probability value, making it easy to derive a multiplicative approximation from an additive approximation). On query type 5 and join queries, interestingly enough, the best-tree strategy is significantly faster than all simple baselines, meaning that our evaluation tree is useful and that different evaluation strategies need to be applied to different nodes of the tree.

Unlike EvalDP, whose performance is linear in the size of the data (and remains in our experiments relatively uniform for all queries), the performance of ProApproX is obviously related to both the size and the complexity of the lineage DNF. The part that is directly related to the size of the data in ProApproX is the XQuery time recorded by BaseX for lineage query evaluation. But the part that is sensitive to the size of the DNF lineage is the compilation time. As shown in Table III, the majority of the time is spent in XQuery evaluation for all queries on this dataset. For queries with larger DNFs, the *Compilation time* becomes relatively important. The time required for the Exploration and Evaluation phases remain negligible in all cases. Finally recall that EvalDP is neither able to deal with join queries, nor with long-distance dependencies, contrarily to ProApproX.

[6]http://www.cs.ox.ac.uk/projects/SPROUT/

[7]We were not able to make the other probability computation functions, the exact *conf()* and the deterministic approximation *conf('M',ε)* based on d-trees, run on relations of arbitrary arity.

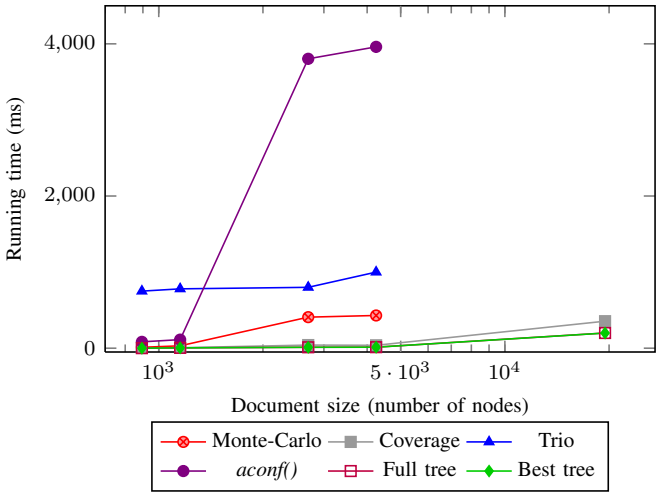


Fig. 5. Running time of the different algorithms on query Q_3 of the movie dataset [2]; times greater than 6s are not shown; times for Trio from [2]

With respect to SPROUT's *aconf()*, we recorded improvements of one order of magnitude on the largest DNFs. For the last categories of queries, runtimes for *aconf()* were of the order of several seconds, while ProApproX never exceeded 100 milliseconds.

Fig. 4 shows how accurate the result of ProApproX are, compared to exact results, on one particular run of the algorithm (since we use sampling, errors tend to be averaged out if we consider multiple runs). It shows that for most results ProApproX achieves high accuracy. For 14 out of 15 queries, we record an accuracy far below our relative error margin of 0.1. For one of the 15 queries, we have an error slightly higher than 0.1, in line with what is expected: with a δ of 95% and for 15 queries, there is a chance of $1 - 0.95^{15} = 53.6\%$ that we produce, at least once, a relative accuracy beyond 0.1.

Movies: For the three queries of this dataset, the run time on the five p-documents was recorded to be almost one order of magnitude lower then the performances presented by the authors in [2] using Trio, a relational probabilistic DBMS [16]. In Fig. 5, we plot the performance over the join query Q_3; points missing in the figure mean runtimes greater than 6 seconds. It is interesting here to note that Trio has a performance of about 40 seconds on the largest p-document (19,475 nodes), while ProApproX took around 200 milliseconds.

Fig. 5 also compares the best-tree strategy to the full-tree strategy (always evaluating the leaves of the evaluation tree) and shows that in this particular example, the latter is always efficient (or very close to the most efficient). It seems also that it is quite efficient to apply the coverage algorithm to the whole formula, which hints that the full tree might not have many levels (the lineage clauses might be highly correlated). We have seen in Fig. 3 that, for some other datasets, the coverage algorithm is not efficient. The point here is that by using a cost optimizer, we are able *in all cases* to find the most efficient evaluation strategy (or one close to it).

Recorded run times for MayBMS were much higher than times spent by the best evaluation tree in ProApproX to compute the probabilities. Furthermore, we noted that for the last query, whose DNF size is of 47,011 variables, and over the largest document (19,475 nodes), SPROUT was running indefinitely, failing to return an estimation of the DNF probability; we stopped the process after two hours. The reason why *aconf()* may perform disappointingly on these DNFs is that it was not really designed for lineages with large clauses such as those we get from PrXML querying. Conversely, ProApproX would probably perform worse than SPROUT on typically relational data and queries.

Finally, note that to estimate the probability of a query, a MayBMS user has to choose among the different probability computation functions; if she calls *aconf()*, she will *always* get approximations of the result, even in cases when the exact computation is efficient, unlike ProApproX. Indeed, when one varies approximation parameters, ProApproX ends up switching to an exact algorithm, while the runtime of MayBMS keeps increasing as the conditions on approximations become stricter.

Synthetic Data: Fig. 6 presents the performance of different evaluation methods on synthetic data: additive Monte-Carlo, self-adjusting coverage, evaluation over a fully compiled tree (Full tree), and evaluation over the best execution plan (Best tree) that ProApproX chooses for a DNF probability computation. Note that times shown on the top line are actually anywhere beyond. The running time of the self-adjusting coverage algorithm is directly proportional to the size of the formula (i.e., appears as an exponential in a log-linear plot). This is not the case, however, for Monte-Carlo sampling because of the use of the lower bound on the DNF probability: when this value is relatively high (apparently the case for most of the big random DNFs of Fig. 6), Monte-Carlo sampling can be very efficient. On the other hand, when the lower bound is low, the Monte-Carlo algorithm can be quite ineffective, as is shown by the irregular variation recorded for this algorithm over DNFs of different sizes. Our best-tree strategy significantly outperforms the self-adjusting coverage algorithm and gives very low running times (of usually less than 100ms) for DNFs holding until 10^4 variables. It also outperforms Monte-Carlo for most of the cases, especially for DNFs with low probabilities. However, when the data scales (DNFs with sizes larger than $\sim 10^{5.6}$), Monte-Carlo may record better time, which is due to the compilation time needed by ProApproX before deciding on the best strategy. Yet the difference in running time remains reasonable. Interestingly, for very large formulas, the best-tree strategy does a better job than the full-tree strategy. Most importantly, the best tree is *always* under 6s, even though it is sometimes not the quickest (cf. Monte-Carlo), which empirically demonstrates that it is much more robust.

IX. Related Work

Probabilistic data management: Managing probabilistic and uncertain data is a topic of much current interest in database research, and there has been a significant amount of work under the relational database scheme. Extensive literature and many systems and prototypes have been introduced, such as Trio [16] and MayBMS [18], [19].

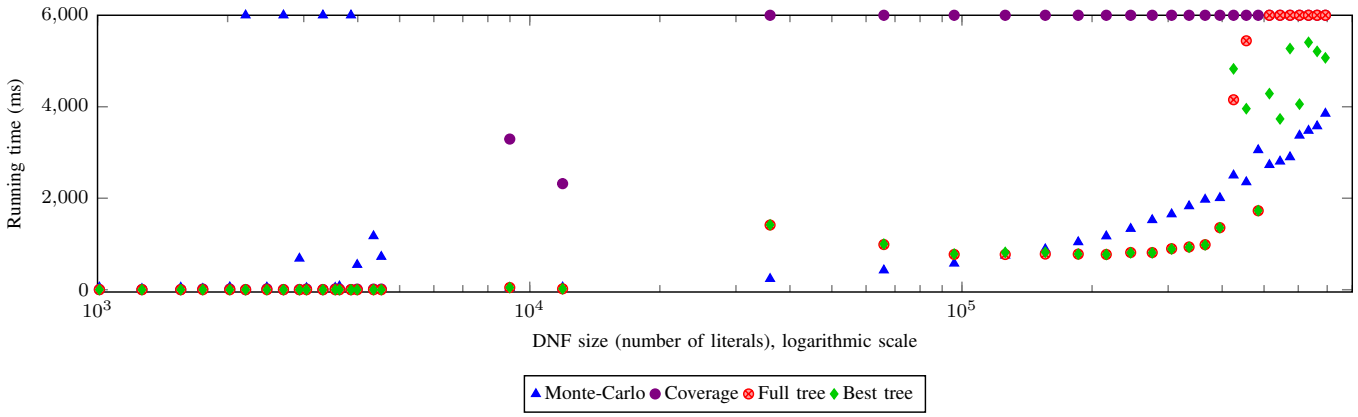


Fig. 6. Running time of the different algorithms on the synthetic dataset

SPROUT [20], the query engine integrated in the probabilistic relational database MayBMS [19], considers query evaluation on probabilistic tables, for a query language that goes from Boolean conjunctive queries without self-joins [21] to positive queries [3], to queries with negation [22]. For tractable queries, the confidence computation is based on OBDDs [23]. If a DNF lineage is captured, it is compiled to a novel kind of decision diagrams called *d-trees*, exploiting negative correlations (inconsistency), independence, and factored representations. In [3], the authors pointed out the need for more advanced and more accurate – but still efficient – techniques for estimating lower and upper bounds of probability values.

Fink and Olteanu [24] also addressed the problem of approximating lineage formulas that are hard to evaluate with read-once formulas, where every variable occurs at most once, or read-once formulas in disjunctive normal form; note that the input lineage formula cannot include negated literals, which makes it inapplicable to our setting.

Probabilistic XML: Meanwhile, the semistructured model has received less attention. See [4] for a survey of the probabilistic XML literature.

Early work on query evaluation over p-documents was carried out by Kimelfeld, Kosharovsky, and Sagiv in [1], [5], resulting in the EvalDP algorithm that natively processes positive tree-pattern queries, without join. The algorithm is restricted to run only over p-documents with local dependencies, i.e., where dependency can only happen between children of a same parent node. This linear-time bottom-up processing of the tree has first-rate performance, and has been generalized to arbitrary monadic second-order queries through tree automata techniques [25], but is not applicable to queries or data that involve correlations.

In [1], Kimelfeld, Kosharovsky, and Sagiv proposed as a proof of the existence of a randomized polynomial-time approximation scheme an alternative sampling algorithm to obtain a multiplicative approximation of a Boolean query. The probability of query Q being true in a random instance $\mathscr{P}$ can be computed [1] by: $\sum_{i=1}^{n} \Pr(m_i \lhd \mathscr{P}) \times \Pr\left(\bigwedge_{j=1}^{i-1} \neg(m_j \lhd \mathscr{P}) \mid m_i \lhd \mathscr{P}\right)$, where $\Pr(m_i \lhd \mathscr{P})$ is the probability that a match m_i of Q remains in $\mathscr{P}$. The first term is easy to compute by just gathering the probabilities of all events involved in the match; the second term can be approximated by conducting biased draws to considerate the probability that none of the preceding matches exists in the random document $\mathscr{P}$, given that a current match m_i appears in that same document. We implemented this algorithm in [26], a demonstration of an early version of ProApproX without any form of cost estimation, and the evaluation leads to very good accuracy. Nevertheless, the convergence guarantee [1] that is obtained from Hoeffding's inequality requires a running time growing in $O(m^3 \ln m)$ in the number of matches (i.e., in the number of clauses of the lineage formula).

DNF probability estimation: We replaced this method by the self-adjusting coverage algorithm, a linear multiplicative estimation of the number of assignments satisfying a DNF, and an extension of the earlier *coverage algorithm* for the *union of sets* problem, proposed by Karp, Luby and Madras [14] (with the background motivation of managing network reliability issues). The DNF counting problem is the special case of the DNF probability problem when all variable probabilities are 0.5, in which case $\Pr(\varphi) = \frac{\#\varphi}{2^N}$ (with $\#\varphi$ the number of satisfying assignments of φ). Both coverage and self-adjusting coverage algorithms for the DNF counting problem can easily be modified for the DNF probability problem, yielding (ε, δ)-approximation algorithms, as shown in a subsequent work by Luby and Velickovic [27], and as deployed in ProApproX.

Systems: At the practical level, a full-fledged probabilistic XML data management system is still missing. Hollander and van Keulen [2] investigate the adequacy of probabilistic relational databases for querying probabilistic semistructured data by mapping XML to relational data. Queries can then be transformed into relational queries and evaluated using a system such as Trio, for which efficient query evaluation algorithms have been developed, both exact and approximate. The downside of the approach is that relational databases do not exploit the specific characteristics of tree-like data encoded into databases, that for instance makes tree-pattern queries over local models tractable [1].

Preliminary ideas leading to this work were presented in [26], [28] though the material presented in Sections IV–VIII, as well as the use of a cost estimator to optimize the running time of

the probability computation, are fully novel. Details that could not fit in this article are available in [15]. A demonstration of the latest version of ProApproX has been presented in [29].

X. Conclusion

We have introduced an original optimizer-like approach to evaluating query results over probabilistic XML. Though some components are specific to this setting (in particular, the translation from XPath queries to XQuery lineage queries), the core of our system aims at solving the very general problem of computing (approximate) probabilities of propositional formulas built out of independent random variables, in an efficient manner. Our formula decomposition technique assumes that the formula is in DNF, but the technique could probably be extended to arbitrary formulas.

The first principle our approach relies on, as well as the main observation from our experiments, is that *the optimal probability evaluation algorithm to use depends on the characteristics of the formula*: if the formula has few variables, go with the naïve algorithm; if it has few clauses, with the sieve algorithm; if the probability is known to be close to 1, with Monte-Carlo sampling; if the formula was obtained from evaluating a tree-like query over a tree-like structure, use a technique such as EvalDP; if nothing else works, use the self-adjusting coverage algorithm. Our cost model captures this. The second principle is that *different algorithms can be used to evaluate the probability of different parts of a formula*. Our formula decomposition technique, and our exploration of possible evaluation trees, makes use of this.

In ProApproX, the production of the lineage from the original p-document and query and the evaluation of the probability of this lineage are fully independent. Each of these problems can thus be optimized independently (using, respectively, the XML query optimization of a native XML DBMS, and our formula probability evaluation technique). However, it is likely that exploiting the structure of the query to obtain lineage that is already factorized would result in even more efficient evaluation. This is an obvious direction for future research.

Other potential improvements of interest would be (i) to refine our cost model with non-linear functions of the size of the formulas, to take into account second-order terms in the complexity of the algorithms; (ii) to keep the same δ's parameter across the whole tree by performing correlated samples in all branches of the tree – this is not trivial either, since we need to keep track of all partial samples performed; (iii) to exploit the fact that most evaluation algorithms are effortlessly parallelizable (with the exception of the self-adjusting coverage algorithm, which requires synchronization) to distribute the probability computation.

Acknowledgment

This work has been partly supported by the European Research Council grant Webdam (under FP7), grant agreement 226513, and by the Dataring project of the French ANR. We would like to thank B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv for providing the EvalDP system and dataset [1], [5], as well as E. Hollander and M. van Keulen for the Movies dataset [2]. We are also grateful to J. Huang and D. Olteanu for their help with MayBMS/SPROUT and for their valuable comments.

References

[1] B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv, "Query evaluation over probabilistic XML," *VLDB J.*, 2009.
[2] E. Hollander and M. van Keulen, "Storing and querying probabilistic XML using a probabilistic relational DBMS," in *MUD*, 2010.
[3] D. Olteanu, J. Huang, and C. Koch, "Approximate confidence computation in probabilistic databases," in *ICDE*, 2010.
[4] B. Kimelfeld and P. Senellart, "Probabilistic XML: Models and complexity," in *Advances in Probabilistic Databases for Uncertain Information Management*, Z. Ma, Ed. Springer-Verlag, 2013, to appear.
[5] B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv, "Query efficiency in probabilistic XML models," in *SIGMOD*, 2008.
[6] S. Abiteboul, B. Kimelfeld, Y. Sagiv, and P. Senellart, "On the expressiveness of probabilistic XML models," *VLDB J.*, 2009.
[7] M. van Keulen and A. de Keijzer, "Qualitative effects of knowledge rules and user feedback in probabilistic data integration," *VLDB J.*, 2009.
[8] T. Abdessalem, M. L. Ba, and P. Senellart, "A probabilistic XML merging tool," in *EDBT*, 2011, demonstration.
[9] B. Kimelfeld and Y. Sagiv, "Matching twigs in probabilistic XML," in *VLDB*, 2007.
[10] P. Senellart and S. Abiteboul, "On the complexity of managing probabilistic XML data," in *PODS*, 2007.
[11] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM J. Comp.*, 1979.
[12] K. D. Heidtmann, "Improved method of inclusion-exclusion applied to k-out-of-n systems," *IEEE Trans. Reliability*, 1982.
[13] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *J. American Statistical Association*, 1963.
[14] R. M. Karp, M. Luby, and N. Madras, "Monte-Carlo approximation algorithms for enumeration problems," *Algorithms J.*, 1989.
[15] A. Souihli, "Querying probabilistic XML," Ph.D. dissertation, Télécom ParisTech, 2012.
[16] M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. D. Sarma, R. Murthy, and T. Sugihara, "Trio-One: Layering uncertainty and lineage on a conventional DBMS," in *CIDR*, 2007.
[17] C. Koch, "MayBMS: A system for managing large uncertain and probabilistic databases," in *Managing and Mining Uncertain Data*, C. C. Aggarwal, Ed. Springer-Verlag, 2009.
[18] L. Antova, C. Koch, and D. Olteanu, "Query language support for incomplete information in the MayBMS system," in *VLDB*, 2007.
[19] J. Huang, L. Antova, C. Koch, and D. Olteanu, "MayBMS: a probabilistic database management system," in *SIGMOD*, 2009.
[20] D. Olteanu, J. Huang, and C. Koch, "SPROUT: Lazy vs. eager query plans for tuple-independent probabilistic databases," in *ICDE*, 2009.
[21] D. Olteanu and J. Huang, "Secondary-storage confidence computation for conjunctive queries with inequalities," in *SIGMOD*, 2009.
[22] R. Fink, D. Olteanu, and S. Rath, "Providing support for full relational algebra in probabilistic databases," in *ICDE*, 2011.
[23] D. Olteanu and J. Huang, "Using OBDDs for efficient query evaluation on probabilistic databases," in *SUM*, 2008.
[24] R. Fink and D. Olteanu, "On the optimal approximation of queries using tractable propositional languages," in *ICDT*, 2011.
[25] S. Cohen, B. Kimelfeld, and Y. Sagiv, "Running tree automata on probabilistic XML," in *PODS*, 2009.
[26] P. Senellart and A. Souihli, "ProApproX: a lightweight approximation query processor over probabilistic trees," in *SIGMOD*, 2011, demonstration.
[27] M. Luby and B. Velickovic, "On deterministic approximation of DNF," *Algorithmica J.*, 1996.
[28] A. Souihli, "Efficient query evaluation over probabilistic XML with long-distance dependencies," in *EDBT/ICDT PhD Workshop*, 2011.
[29] A. Souihli and P. Senellart, "Demonstrating ProApproX 2.0: a predictive query engine for probabilistic XML," in *CIKM*, 2012, demonstration.