Querying probabilistic XML

Asma Souihli

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2012-ENST-0046

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

« Informatique »

présentée et soutenue publiquement par

Asma SOUHILI

Le 21 Septembre 2012

Interrogation des données XML probabilistes

Querying Probabilistic XML

Directeur de thèse : **Pierre SENELLART**

Jury

Mme Angela BONIFATI, Professeur Univ. Lille 1 (rapporteur)
M. Gerome MIKLAU, Professeur Univ. Massachusetts (rapporteur)
M. Talel Abdessalem, Maître de Conférences Télécom ParisTech
M. Philippe RIGAUX, Professeur CNAM
M. Reza AKBARINIA, Chercheur INRIA Sophia Antipolis

THÈSE

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

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 dissertation efficient algorithms for evaluating tree-pattern queries with joins over probabilistic XML or, more specifically, for approximating the probability of each item of a query result. The approach relies on, first, extracting the query lineage over the probabilistic XML document, and, second, looking for an optimal strategy to approximate the probability of the propositional lineage formula.

ProApproX is the probabilistic query manager for probabilistic XML presented in this thesis. The system allows users to query uncertain tree-structured data in the form of probabilistic XML documents. It integrates a query engine that searches for an optimal strategy to evaluate the probability of the query lineage. ProApproX relies on a query-optimizer–like approach: exploring different evaluation plans for different parts of the formula and predicting 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 a number of most popular previous probabilistic XML querying works, as well as on synthetic data.

An early version of the system was demonstrated at the ACM SIGMOD 2011 conference. First steps towards the new query solution were discussed in an EDBT/ICDT PhD Workshop paper (2011). A fully redesigned version that implements the techniques and studies shared in the present thesis, is published as a demonstration at CIKM 2012. Our contributions are also part of an IEEE ICDE 2013 research paper submission.

Remerciements

Pierre Senellart est une personne exceptionnelle, ma décision de faire une thèse de doctorat était en grande partie motivée par la possibilité que j'espérais tant avoir de travailler sous sa direction, je portais une grande admiration à son professionnalisme et la passion qu'il portait pour son travail de chercheur. Je tiens tout d'abord à le remercier de m'avoir fait confiance malgré les connaissances plutôt légères que j'avais en octobre 2009 sur les bases de données probabilistes, puis pour m'avoir guidé, encouragé, conseillé, fait beaucoup voyager pendant les trois ans tout en me laissant une grande liberté et en me faisant l'honneur de me déléguer plusieurs responsabilités dont j'espère avoir été à la hauteur. Pierre m'a enseigné l'exemple à travers la gentillesse et la patience qu'il a manifestées à mon égard durant cette thèse, malgré son emploi chargé et le nombre de projets qu'il devait gérer en parallèle. Je suis très fière d'avoir été une de ses premiers disciples durant sa carrière d'enseignant chercheur à Télécom ParisTech, et ne sais comment exprimer la reconnaissance que je portais tout les jours autrement qu'en lui promettant d'agir de la sorte avec des étudiants ou employés dans ma situation, si un jour l'occasion m'en serait donnée.

Cette thèse, aboutissement de longues années d'études, je la dois beaucoup à mes chers parents, qui ont fait ce que je suis devenue aujourd'hui ; pour leur immense dévouement et leur incroyable présence, je leur dédie cet accomplissement et leur fait la plus grande part de ma joie. À mon frère si merveilleux, mon ami et mon exemple. Il m'est impossible de trouver des mots pour dire à quel point je suis fière d'eux, et à quel point je les aime.

J'aimerai remercier Gerome Miklau et Angela Bonifati d'avoir accepté avec grand plaisir d'examiner ma thèse, ainsi que tous les membres du jury pour avoir répondu à l'invitation. Je remercie Bogdan Cautis et Talel Abdessalem de m'avoir permis d'effectuer mon stage de Mastère au département INFRES en début de l'année 2009 et de faire partie de la famille DBWeb. Je tiens aussi à mentionner le plaisir que j'ai eu à échanger des idées au sein de l'LIRMM, malgré le court séjour à Montpellier, et j'en remercie ici Reza Akbarinia de m'avoir fait l'honneur de participer au Jury de soutenance. Un remerciement particulier à Ahmed Serhrouchni d'avoir été apprêté à me porter de l'aide quand j'en avais

besoin, avec une confiance qui m'a beaucoup touchée. Ceci n'est pas étranger à sa bien-connue bonté et sa remarquable serviabilité.

Merci à Elie pour sa belle amitié et sa présence, pour sa disponibilité et son écoute, et pour les débats constructifs qui m'ont énormément aidé pendant les débuts de ma thèse. Je lui souhaite la réussite et un succès distingué dans la sienne. Pour conclure, je souhaite remercier tous mes amis de Télécom ParisTech, avec lesquels j'ai beaucoup échangé et passé de très beaux moments. Merci aussi à Silviu pour son apport grâce auquel de nombreuses idées clés du Chapitre 4 ont pu être élaborées. Merci à Imen, Nora, Nessrine, Mike, Sylvain, Fabian, Khaled, Marilena, Simon, Hayette, Fabien et tous les autres, la liste est loin d'être exhaustive, qu'ils m'excusent ! C'était un plaisir de passer cette belle période de ma fin de vie estudiantine en votre compagnie !

Acknowledgements

Pierre Senellart is an exceptional person, my decision to do a PhD thesis was largely motivated by the possibility I could hopefully have to work under his supervision, I wore a great admiration for his high proficiency and the passion he shows for his work. Pierre was demonstrating a very noble quality through the kindness and patience he manifested towards me during this thesis, despite his busy schedule and the number of projects he had to manage simultaneously. First, I want to thank him for his trust despite my little knowledge in October 2009 on probabilistic databases, for his guidance, support, advice, and for making me travel a lot during the past three years while offering me a lot of freedom and the opportunity to manage a number of responsibilities. I hope I delivered on the trust he placed in me and that I did well. I am very proud to have been one of his first disciples in his career and do not know how to express the gratitude I carried every day other than promising him to do so with my students or employees, if one day the opportunity is given to me.

This graduation, a culmination of many years of study, I am indebted to my dear parents for their amazing devotion and incredible presence; for them I am dedicating this achievement. To my wonderful brother and precious friend. It is impossible to find words to tell how much I am proud of them and how much I love them.

I would like to thank the members of my thesis jury and especially Gerome Miklau et Angela Bonifati, who accepted to take the time to review my thesis. I want to acknowledge Bogdan Cautis and Talel Abdessalem for offering me the opportunity to do my Master internship at the INFRES department in the beginning of 2009, and for allowing me to be part of the DBWeb family. I also want to mention the pleasure I had to exchange ideas within the LIRMM, despite the short stay in Montpellier, and I thank Reza Akbarinia here for giving me the honor to participate in the Jury of defense. I owe thanks to Pr. Ahmed Serhrouchni who have been primed to bring me help when I needed it, with a trust that moved me a lot. I am always grateful for this.

I wish to thank Elie for his beautiful friendship and presence, and for the constructive discussions that helped me a lot during the beginnings of my research.

I wish him all the success in his PhD. Ultimately, I am grateful to all my friends at Telecom ParisTech, with whom I shared a lot and had a very nice time. Let me acknowledge Silviu for the productive debates we had and that raised a number of key ideas in Chapter 4. Thanks to Imen, Nora, Nessrine, Mike, Sylvain, Fabian, Khaled, Marilena, Hayette, Fabien. Please excuse the omissions that I am bound to have made in such a list! It was a real pleasure to spend the last part of my student life with you!

Contents

List of Figures

List of Tables

Introduction

Many applications, such as uncertain data integration [57], XML warehousing [50], uncertain version control systems [1], or Web information extraction [2, 24, 51], require uncertainty management. In this latter example, the system could have some confidence in the information extracted, that can be a probability of the information being true (e.g., conditional random fields [55]) or an ad-hoc numeric confidence score. 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 adequate, 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 [37] in particular.

Probabilistic documents or *p-documents* [4, 34] 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 but exponentially large set of *possible worlds*, each with a particular probability). Given a probabilistic document, we propose in this thesis to investigate how to efficiently query it. We are interested in querying probabilistic XML documents, using XPath, a query language that provides the ability to navigate around the XML tree and select nodes by a variety of criteria and predicates. In what follows, we discuss a solution for dealing effectively with a range of XPath queries, defined by the *tree-pattern* queries, possibly with *joins* (Section B.2). Given a probabilistic XML document, the confidence related to a match y of a query over the current probabilistic document, is the combined probability of all possible worlds in which the match y occurs. This computation is the key operation that characterizes probabilistic databases over traditional databases. However, computing the exact probability of an answer occurring in a query result over a p-document is #P-hard [35]. Under data complexity, there are fully polynomial-time randomized approximation schemes [35] to estimate the

confidence of a given result to a query Q: we thus focus on efficiently approximating this quantity. 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 [35, 50], we reduce the problem of approximating the probability of 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 matches to the query constitutes the propositional *lineage* φ of the query. 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 [56].

In contrast with existing work [34–36] 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* [50] queries) and aim at a practical solution for querying p-documents. ProApproX is a query engine that implements a number of solutions discussed in this thesis. The system 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.

To summarize, we highlight the major contributions of our thesis, through which ProApproX reveals its originality:

1. The support of a broader range of XPath queries over a more general data model than current probabilistic XML systems (Chapter 1);

2. A cost model for a variety of probability evaluation algorithms, based on which the selection of the most efficient algorithm is performed (Chapter 3);

3. Simplification of the computation process via a decomposition of the probabilistic lineage into independent computational cells (Section 4.1);

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 4.2);

5. An exploration of the space of evaluation plans based on the proposed cost model (Section 4.3);

6. A practical, implemented system, with demonstration interface, that represents a main contribution of the thesis (Chapter 5).

Chapter 1 also introduces the background knowledge by presenting probabilistic XML models, syntax, semantics of the *tree-pattern* query classes supported in ProApproX, the probabilistic lineage, and a proper statement of the challenge behind processing queries over probabilistic trees. Chapter 2 presents a number of relevant state-of-the-art systems and solutions in querying probabilistic data, both over the relational and semistructured model (we focus though on the confidence computation techniques deployed in these systems). Before concluding, a last chapter presents an extensive experimental evaluation of the proposed system, and compares the performance of our query engine with the state of the art.

Chapter 1

Preliminaries

1.1 Probabilistic XML

We recall here some basic notions about probabilistic XML (or PrXML) as an uncertain data model. The model presented here was introduced and studied in [4, 34, 35] as a generalization of previously existing models [6, 43, 58]. 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.

1.1.1 Probabilistic XML Models

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 [43, 58] (named $\mathsf{PrXML}^{mux,ind}$ in [4]), 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. Figure B.1 illustrates a probabilistic tree of the local dependency model. This example can be a result of an integration of a number of directories, containing information about instances of *persons*. For example, confidences may depict the gathered statistics when looking for an information out of a given set of integrated directories. A first phone number `3333` related to the instance person named `Chris` was found with an independent chance or probability from other nodes, having the value of 0.1, or 10%. The independency of this distribution is indicated by the fictive

node *ind* in the abstraction of PrXML^{mux,ind}. The *mux* node in the same example expresses a mutual exclusiveness between its different child nodes, holding in this example instances of possible addresses found for `Chris`.

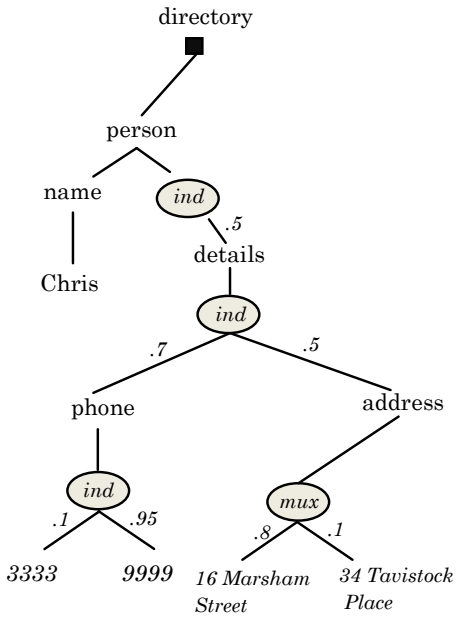


Figure 1.1: PrXML^{mux,ind} - *local dependency* model

- In the *long-distance dependency* model [6] (PrXML^{cie} in [4]), 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 example PrXML^{cie} of Figure B.2 has the power to express the same conditions used in the PrXML^{mux,ind} of Figure B.1, by using independent Boolean variable mapped with their probabilities. The mutual exclusiveness between address instances is henceforth expressed through propositional formulas involving Boolean variable and their negation. Furthermore, dependency between distant `city` nodes in this tree, is possible through the use of a same variable. This may give utterance to a generalized semantic that unifies the statistic related to this instance over the whole database.

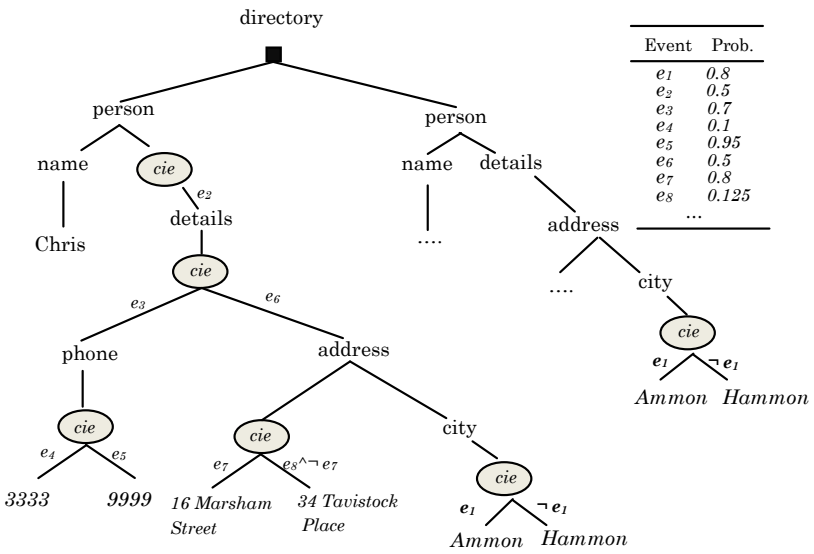


Figure 1.2: PrXML^{cie} - *long-distance dependency* model

1.1.2 From PrXML^{mux,ind} to PrXML^{cie}

As noted in [4], local dependencies expressed by *ind* (independent) or *mux* (mutually exclusive) nodes can be tractably translated into *cie* nodes (Figure B.3). The probability related to the node $Child_3$ is mapped to the boolean variable e_2 in the equivalent PrXML^{cie} tree. $Child_1$ and $Child_2$ are mutually exclusive in the PrXML^{mux,ind} representation, through the introduction of a parent node *mux*. the translation is performed by building two inconsistent propositional formulas, having respectively probabilities of 0.3 and 0.5. It suffices in this simple case to create a new Boolean variable e_1 having the probability 0.3, and link it to the $Child_1$ node; then find a formula for $Child_2$ having the probability 0.5 and containing a negation of e_1.

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 work presented in this thesis.

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 (Figure B.4). 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

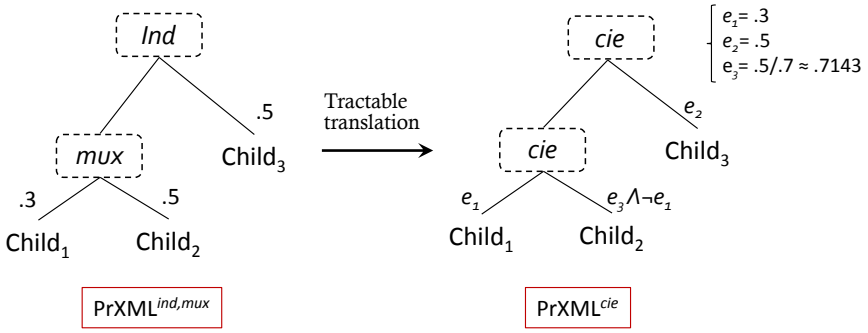


Figure 1.3: Tractable translation: from $\mathsf{PrXML}^{mux,ind}$ to PrXML^{cie}

document. The probability of this document is that of all truth assignments that generate it.

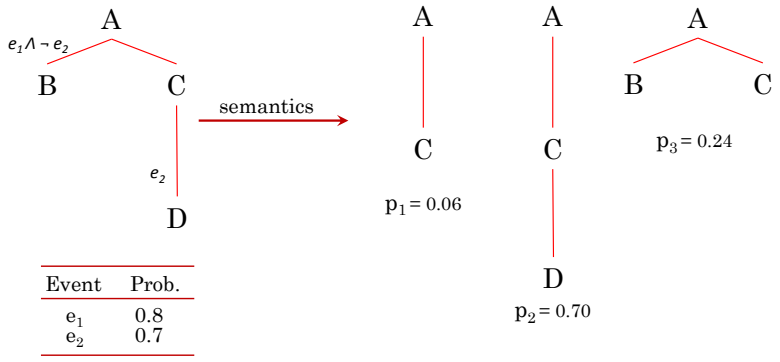
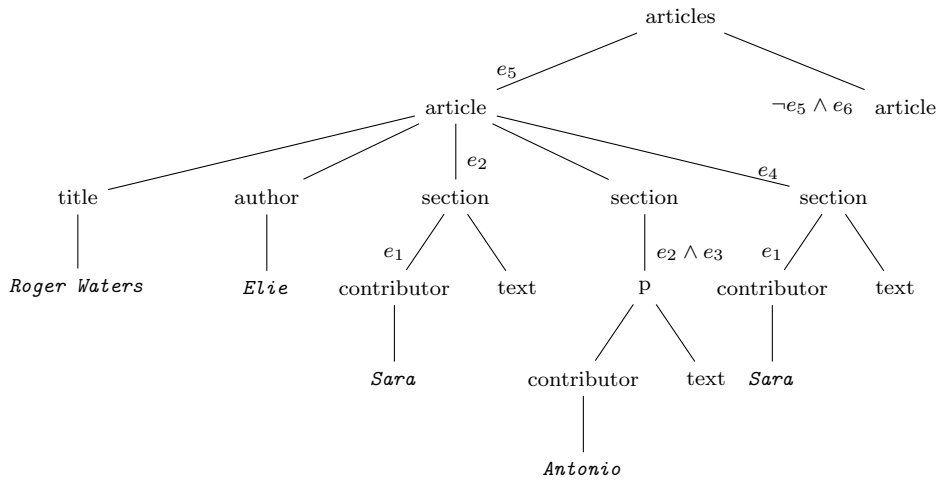


Figure 1.4: Unfolding a PrXML^{cie} tree: possible worlds

As an example, Figure B.5 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 [1]. For ease of presentation, *cie* nodes are shown by simply annotating edges with conjunctions of literals. For example, the first and third "contributor" elements in the tree appear under *cie* nodes that retain them only if e_1 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 [1], with the probability distribution of the $\Pr(e_i)$'s typically inferred by a trust inference algorithm [40].



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
...	

Figure 1.5: Tree representation of a p-document with long-distance dependencies

1.2 Querying Probabilistic XML

Choosing between a more concise PrXML model with strong expressive power, and efficiency of query evaluation is clearly and naturally a tradeoff. The authors of *EvalDP*, a linear-time bottom-up algorithm for evaluating simple tree pattern queries on local dependency p-documents [35], state that:

> "Queries have an efficient (exact) evaluation only when distributional nodes are probabilistically independent."

Tree-pattern query answering over $\mathsf{PrXML}^{mux,ind}$ is linear-time [35], and (as will be explained later) intractable over cie nodes. In this dissertation, we choose to study the complexity of a specific class of XPath queries (more general than tree-pattern queries) over PrXML^{cie}, without a restriction on nodes-dependency over the tree, and analyze the main challenges behind a convenient solution that aims for practicality, with a good accuracy of results.

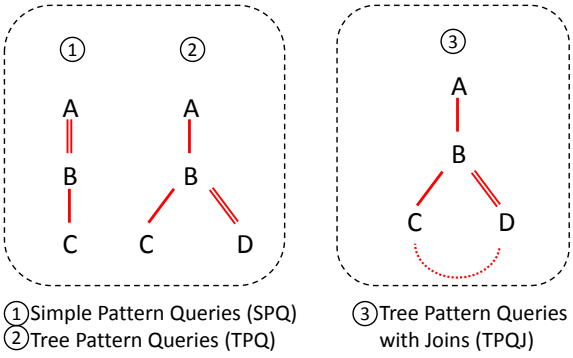


Figure 1.6: Types of queries supported (ProApproX)

1.2.1 Supported Queries

The query language we consider in this work is *tree-pattern queries with joins* [6]. 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 [6], 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 Figure B.5, 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* [50] queries, that includes, for example, the whole positive fragment of first-order XPath [12].

1.2.2 Probabilistic XML Query Lineage: Boolean Projection of the Query

A Boolean query over a deterministic document returns either true or false. A Boolean query Q over a p-document $\mathscr{P}$ returns the *probability* that Q is true in $\mathscr{P}$, that is, the sum of the probabilities of all possible documents of $\mathscr{P}$ where Q is true. A fundamental observation [50], due to the locally monotone nature of the query language, is that the probability of a query over a p-document is the probability that one of the matches of the query over the underlying deterministic document remains in $\mathscr{P}$. We will use this fact to transform our probabilistic XML querying problem into a probability computation problem over a propositional formula in disjunctive normal form (DNF).

Recall the example query Q_1 that looks for the list of contributors to the revisions of the article entitled "Roger Waters". Looking at Figure B.5 we can see three matches to Q_1 on the deterministic document underlying the p-document, corresponding to the three different "contributor" nodes. For each match, we can construct the conjunction of all probabilistic literals on the path from the root to one of the nodes matched by the query:

$$c_1 = e_5 \wedge e_1 \wedge e_2$$
$$c_2 = e_5 \wedge e_2 \wedge e_3$$
$$c_3 = e_5 \wedge e_1 \wedge e_4$$

These conjunctions are called the *probabilistic lineage* of each match.

As noted, the probability of a locally monotone query Q over a p-document is exactly the probability of the disjunction of probabilistic lineages of all matches to the query on the underlying deterministic document. The probability of Q_1 over the p-document of Figure B.5 is thus the probability of the following propositional formula, called the *probabilistic lineage* φ_1 of the Boolean query Q_1 over this document:

$$\varphi_1 = c_1 \vee c_2 \vee c_3 = (e_5 \wedge e_1 \wedge e_2) \vee (e_5 \wedge e_2 \wedge e_3) \vee (e_5 \wedge e_1 \wedge e_4).$$

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 of a regular XML document stored and managed by an ordinary native XML DBMS. The name of this attribute can be specified by the user, and for databases over which we run the experiments, the attribute was named "event".

To retrieve the lineage of a query, we need to transform our XPath query Q into an XQuery query Q' that returns the concatenation of all "prob" attributes of each match. In our example, Q_1 is transformed into query Q'_1:

```
for $a in //article
for $b in $a/title/text()[.='Roger Waters']
for $c in $a//contributor
let $leaves:=($b,$c)
let $atts:=(for $i in $leaves
  return $i/ancestor-or-self::*/@event)
return <match> {
  distinct-values(
    for $att in $atts
    return string($att))} </match>
```

A generic translator is implemented as part of ProApproX to transform a tree-pattern query with joins encoded in XPath into an XQuery lineage query.

We have then separated the problem of querying probabilistic XML into two independent problems:

- evaluating an XQuery query over a regular XML document;
- computing the probability of a formula in DNF.

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 (Chapter 5). The latter problem is the focus of Chapter 4.

1.2.3 Probabilistic XML Query Lineage: Set of Answers

The query about contributors has three matching paths in the example tree of Figure B.5, but has to return only two distinguished answers to the user. The first answer is about the contributor "*Sara*" that has two instances over the tree, and a corresponding DNF lineage of:

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

[1] http://basex.org/

the remaining item, "*Antonio*", appears once and has thus the probability of the DNF lineage composed only of the clause c_2:

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

A second translated query for extracting items' lineages is built mainly through adding a grouping function to the lineage query Q'. This would return the set of items that match the user query Q, mapped with their corresponding DNF lineage.

Items lineage query Q'':

```
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 $i in $leaves
   where $x1=$val (: The grouping function :)

     return $i/ancestor-or-self::*/@event)
     return <clause>{distinct-values
     (for $att in $atts return string($att))
     }</clause>}
  </match>
```

1.2.4 Complexity of Query Evaluation: Problem Statement

It is easy to see that any DNF formula can be generated as the lineage of even a trivial XPath query such as `//A` [34]. The probability p of a propositional formula φ in DNF denotes the probability that a random, independent, truth assignment to the variables (or events) succeeds to satisfy the formula φ. If there are n Boolean variables, then there are 2^n possible assignments.

The problem of counting the exact number of solutions to a DNF or its probability is known to be $\#P$-hard in general, even when all events have the same probability [56]. This implies in particular that there is no polynomial time algorithm for the exact solution if $P \neq NP$. Note that $\#P$ is an intractable complexity class which contains NP [56].

In the context of a p-document of PrXML^{cie}, this enumeration of assignments is conceptually equivalent to an evaluation of queries in each possible world or

ordinary document, of the p-document following the distribution of events. The sum of probabilities of possible worlds $\{\mathscr{P}_1, ..., \mathscr{P}_n\}$ that satisfy the query (i.e., a clause in the probabilistic lineage DNF is satisfied), is the probability value that answers the stated problem.

However, evaluation of DNF probability is commonly resolved by an approximation scheme, especially when an estimation of this quantity is sufficient in practice. For example, an additive approximation following a basic Monte Carlo sampling can be very effective and leads to good accuracy. Nevertheless, the naïve Monte Carlo performs badly for the case where the number of satisfying assignments to a DNF is exponentially small, with respect to the total (huge) number of possible assignments. From an approximation point of view, we would need exponentially many samples from the set of possible assignments to get a given number of successes.

Instead of picking assignments uniformly at random and testing each clause, we could sample from the set of assignments that satisfy at least one clause (but not uniformly). Many strategies for *sampling only the important space* were introduced from which we cite the algorithms of Karp, Luby, and Madras in [32]. Also, when the number of literals per clause is constant, Ajtai and Wigderson [7] provide an $(\varepsilon, \delta = 0)$ approximation algorithm that has a run time polynomial in the approximation parameters and the input DNF length. Given a tolerated error ε on the result, and a reliability factor δ for the approximation, we can thus find in the literature a number of proposed fully-polynomial randomized approximation schemes, or FPRAS, for the DNF probability problem. Still, a FPRAS will always have a run time polynomial in the DNF size, the error $(1/\varepsilon)$, and, the reliability factor $(\log 1/\delta)$. This fact leaves the problem open for big DNFs, where highly correlated clauses may even break out more difficulties.

Since the issue is fundamentally hard, we aim at designing optimized solutions for exact or approximate query answering. The main idea starts from approaching a hard DNF by performing a number of simplifications. These operations give result to a decomposition of the initial DNF lineage into several independent sub-formulas. We then introduce a number of options to evaluate the probability of the decomposition, increasing then the chance of targeting the less expensive evaluation plan. We discuss in the rest of our thesis how it is possible to optimize the probability computation, via a prospective approach that relies on estimating costs of different possible computation plans.

Chapter 2

Related Work

2.1 Relational Probabilistic Databases

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 setting. Extensive literature around (especially) relational probabilistic databases, was also introduced. Among the earliest representation models, we cite the work of Barbará et al. [10], where probability attributes are allocated for uncertain tuples. Ré, Dalvi, and Suciu [48] presented in 2006 the block-independent model (fairly analogous to $\mathsf{PrXML}^{mux,ind}$ in the semi-structured setting), where mutually exclusive tuples of the same relation are grouped inside a given block, as shown in the following abstract representation [33]:

id	name	age	prob
1	Peter	31	0.2
		30	0.6
		34	0.1
2	Noel	20	0.5
		22	0.2

This relation stores different mutually-exclusive instances of ages related to Peter and Noel: the probability that Peter is 31 is 0.2, the probability that he is 30 is 0.6, etc.

Concerning query evaluations, most of the algorithms have been designed for models where the dependency between the tuples is restricted, i.e., where the probability of each tuple is given, and where the existence of a tuple is independent of the existence of the other tuples [13, 18, 35, 42, 47, 48, 48, 49, 53].

Further interest on dealing with more expressive relational models (closer in the spirit to PrXML^{cie}) led to more complex database management systems

and prototypes. We cite in particular MystiQ [13], Trio [42] where the adopted model allows correlations within tuples of a given relation only, Orion (previously known as U-DBMS) [53], and MayBMS [9, 29]. Most of these systems suggest lineage-oriented query solutions, implementing a range of techniques for computing or approximating probabilistic lineages of queries in less restricted databases (exact evaluation techniques are usually only tractable only if tuples are independent of one another).

In [19], Dalvi and Suciu present a detailed study on the complexity of queries on relational probabilistic databases, and state that even simple conjunctive queries over a tuple-independent database can be #P-hard. This fact contrasts with the situation in probabilistic XML, where Kimelfeld, Kosharovsky, and Sagiv in [34,35] introduced an exact evaluation of simple tree-pattern queries over p-documents with local dependencies, that has a run time linear in the size of data.

MayBMS [38] is an extension of PostgreSQL implementing the conceptual probabilistic model where a probabilistic database is seen as a finite set of database instances of a probabilistic database schema (called possible worlds). Each world has a weight (called probability) between 0 and 1 and the weights of all worlds sum up to 1. Physically, MayBMS uses a purely relational representation system for probabilistic databases called *U-relational databases*, which is based on probabilistic versions of the classical conditional tables (*c-tables*) of the database literature [30]. Each tuple of a U-table has a probability expressed by an atomic condition (using one Boolean variable), or by a propositional formula. the set of variables is *finite*, and each variable is mapped to a given distribution. Random variables are assumed independent in the current MayBMS system. In the coming section, we are going to present fundamental outlines behind the query evaluation techniques implemented in MayBMS.

2.1.1 SPROUT in MayBMS

SPROUT [28], the query engine integrated in the probabilistic relational database MayBMS [29], deals with arbitrary queries over arbitrary U-relations. However, evaluations may have efficient performance for Boolean conjunctive queries with inequalities [45] but without self-joins [18] over tuple-independent U-relations (i.e., restricted databases [46]). The authors of SPROUT introduce more efficient solution for evaluating hierarchical queries over tuple-independent databases, where they exploit regularities in lineages related to this class of queries. The confidence computation is some times based on OBDDs [44], but also on a novel kind of decision diagrams called *d-trees* for harder lineages, with a possible use of approximate computations. 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.

Main Features

The design of SPROUT, as presented by Jiewen Huang in [28], relies on two techniques for probability computation that target different classes of queries. Confidence computation is based on OBDDs [44] for conjunctive queries without self-joins and with tuples inequalities in tuple independent probabilistic databases. Constructing OBDDs of polynomial size to evaluate an arbitrary propositional formula is an NP-hard problem [41]. The authors demonstrate though that in practice, succinct OBDDs can be obtained for the supported classes of queries.

For less restricted queries in a more general data model (i.e., allowing dependency between tuples), and considering only DNF lineages, the formula is compiled into a second kind of decision diagrams, called *d-trees* [47], exploiting negative correlations (inconsistency), independence and applying Shannon expansion. Note however that applying Shannon expansion may only be relevant when formulas hold *few* dependencies. When the DNF is compiled into a complete d-tree, the probability is evaluated exactly. For harder DNFs, the authors devise a solution that uses approximation algorithms to compute the result.

It might be relevant to the purpose of this thesis to briefly explain the principles of approximation techniques in SPROUT: Figure 2.1 sketches a decomposed DNF into a d-tree, where leaves hold parts of the decomposed hard DNF. These leaves will be approximated to return a lower and upper bound of their probability. Let us make an example requirement of an error precision $\varepsilon = 0.005$ and $\delta = 0.95$. SPROUT tests at regular intervals, whether the reached approximation, aggregated at the root level, satisfies the precision requirements, in which case it stops approximating leaves.

Recall the example d-tree of Figure 2.1, we would like to know at this stage if we can stop the computation, return the probability of the query Q and say that the tree is in its final state. At the root level, if the difference between the lower and upper bounds is less than or equal to the error tolerated by the user, the process will end and the result is displayed.

We illustrate the propagation of the result up to the root; we reproduce the notations from [28], [47] for the *independent conjunction* $\odot$ and the *mutually exclusive disjunction* $\oplus$, that link nodes in this example:
$L = 0.3 \oplus (0.35 \odot 0.1991) \oplus 0.548 = 0.917685$
$U = 0.3 \oplus (0.35 \odot 0.2009) \oplus 0.552 = 0.922315$
$U - L = 0.922315 - 0.917685 = 0.00463$ The condition $0.00463 \leqslant (2 \times 0.005)$ is satisfied, thus SPROUT stops and returns the final result.

The authors state that only two simple ways are implemented for computing the upper and lower bounds for DNFs, namely a combination of the Karp-Luby unbiased estimator for DNF counting problem [32] in a modified version adapted

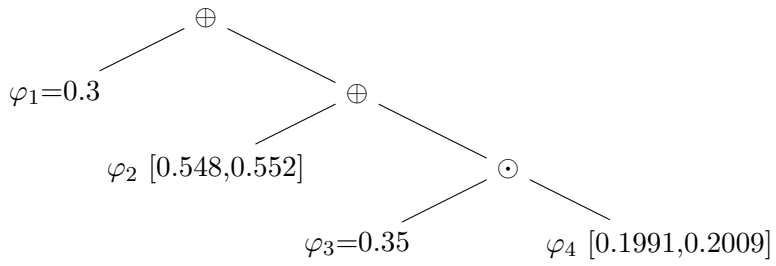


Figure 2.1: A final state of a compilation tree for a probabilistic query Q in SPROUT [28]

to probability computation, and the Dagum–Karp–Luby–Ross optimal algorithm for Monte Carlo estimation [17].

> "Although d-tree outperform state-of-the-art techniques with these approaches, more advanced and more accurate but still efficient techniques for bound estimation are highly desired [28]."

Not so many options are thus proposed for targeting the *best* approximation of a node, and more importantly, there is no investigation regarding the choice of a *most appropriate* algorithm for a given node. When testing some queries on MayBMS, we observed that the stated limits had actually a real impact on performances of the system. In particular, for stricter error bounds, SPROUT leads to a prominent expensive processing. We will show in this thesis that one of our chief aims, among others, is to overcome this situation.

Further Work

Later work by R. Fink and D. Olteanu [22] on SPROUT employs novel exact and approximate knowledge compilation techniques that compute probabilities of events for results of arbitrary relational algebra queries, where the lineage has the shape of an arbitrary propositional formula. This result is explained by the fact that the extended data model, though always assuming independency between tuples, introduces the possibility to express the probability of a tuple using any form of conjunction between the set of Boolean variables in the probability attribute field (i.e., any arbitrary propositional formula).

The authors rely on an event-decomposition framework, similar in spirit to d-trees [47], that is, an incremental compilation algorithm that repeatedly decomposes a propositional formula φ into sub-formulas such that lower and upper bounds of the probability of φ can be efficiently computed from bounds for the sub-formulas. The authors also addressed the problem of approximating lineage formulas that

are hard to evaluate with *read-once* formulas, where every variable occurs at most once (*1OF*), or read-once formulas in disjunctive normal form. The algorithm is run for a given time budget or until the desired approximation is reached [22].

Later on, in [23], Fink, Olteanu, and Rath syntactically define a practical class of tractable relational algebra queries that hold negation and self-joins, and provide an additional algorithm for computing the exact probability of lineages for these queries using 1OF decomposition, and an additional technique inspired by OBDDs, named *mask computation*. Basically, they combine Shannon expansions, mask and 1OF computations to evaluate exactly the probability of the lineage to the query. However, the tractability of these techniques is highly challenged by the hardness of the propositional formula, i.e., level of dependency (overlapping) between clauses of the events lineage.

In a second stage, the authors propose also to introduce mask computations and 1OF decompositions for producing lower and upper bound when approximating *hard* query lineages. In SPROUT, approximations are performed over d-trees, whose design is appropriate to probabilistic lineages in disjunctive normal form. When looking for bounds to the DNF, a lower bound can then be the probability of any pairwise-independent set (leaf) of the *d-tree*, and an upper bound is the sum of probabilities of these partitions (truncated to 1 if greater).

Actually, if DNF lineages are a natural result of positive queries over tuple-independent relations, queries with negation produce a lineage that may require an exponential blowup in the size of the input database when trying to unfold them into DNFs. This observation motivated the design of new approach in [23] for approximating the lineage of arbitrary nested formulas, that uses the new features (mask and 1OF computations) to produce upper and lower bounds for the new type of propositional formula. Nevertheless, it is obvious in [23], that this technique produces different possible bounds, following a choice on variables to consider in the spotted methodology.

Here, we might want to question the stability of approximation schemes in MayBMS. It is clear that starting from *better* bounds may lead to less expensive approximation. Random choice of initial bounds does not allow an estimation of the cost related to the complete approximation process, until the desired precision is reached. Starting also from better bounds will undoubtedly lead better results in case the algorithm is run for a given time budget. In the present thesis, we show that our *cost-model-based* strategy for approximations (Section 3) was motivated by a need for building not only an unvarying approximation scheme, but also one that will be the *best* out of the set of possible execution plans, which will always be used for evaluating the lineage probability.

The authors of [23] also take advantage of some regularities in lineages for

specific classes of queries, by performing particular factorizations and rewriting, that turn out to be beneficial for the probability computation later on.

It is interesting to point out here that techniques of [22] are only applicable for queries without self-joins. This is completely inapplicable to the natural setting of XML databases, for which a lot of self-joins are naturally made (on the Child relation encoding a tree) while processing tree-pattern queries. This observation indicates that SPROUT might have difficulties in evaluating queries over an encoded XML database into relations, as these queries will clearly hold self-joins.

2.2 Probabilistic XML

Meanwhile, the semistructured model has received less attention. An in-depth study on the compared expressiveness and efficiency of probabilistic XML models was elaborated by Abiteboul, Kimelfeld, Sagiv, and Senellart in [4]. Subsequent research has extended the model with continuous probabilistic distributions [3], constraints [15], or possible trees of unbounded size [11].

2.2.1 EvalDP

Early work on query evaluation over p-documents was introduced by Kimelfeld, Kosharovsky, and Sagiv in [34, 35] producing 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.

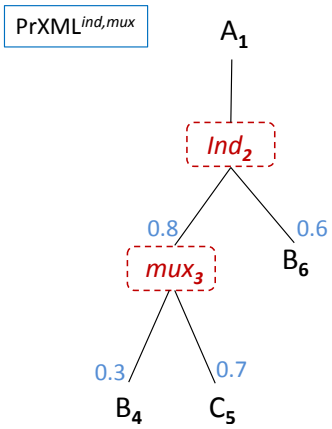
We illustrate the probability computation in EvalDP, via the example $\mathsf{PrXML}^{mux,ind}$ tree of Figure 2.2, and a tree pattern query Q: `/A//B` that looks for a node `B` under an ancestor `A`.

Following a bottom up processing, the algorithm begins by building the set $\mathcal{S} = \{q_1, ..., q_n\}$ that contains all sub-queries of Q, where $q_1 \subseteq q_2 \subseteq ... \subseteq q_n$. It then evaluates each sub-query q_i following the inclusion ordering (starting from the smallest sub-query), through a bottom up traversal of the tree. Computing the result to a given sub-query is done with respect to previous results at previous nodes, i.e., incrementally:

1. At a given node, for example ind_2 (Figure 2.2), and for a given sub-query q_1 : `/B` if the first step node of the XPath sub-query matches the current node, then the evaluation of the sub-query q_1 on the ind_2 is equal to the combined probability of descendent nodes where the the sub-query had a *non-zero* probability result;

2. The computing is performed following the logical operators that link the different probabilistic nodes, and following p-time plans for the local dependency model. The probability between mutual exclusive nodes (linked by a *mux* node) is a result of a convex sum of their respective probabilities. The computing follows an inclusion-exclusion algorithm to compute probability of nodes of an *ind* parent. The result of the probability of q_1 at the ind_2 node is then computed as follows:

$$Pr(ind_2 \models \texttt{/B}) = 1 - (1 - 0.8 \times Pr(mux_3 \models \texttt{/B})) \times (1 - 0.6 \times Pr(B_6 \models \texttt{/B}))$$
$$= 1 - (1 - 0.8 \times 0.3) \times (1 - 0.6) = 0.696$$



	A_1	Ind_2	mux_3	B_4	C_5	B_6
/B	0	0.696	0.3	1	0	1
//B	0.696	0.696	0.3	1	0	1
/A//B	**0.696**	0	0	0	0	0

Figure 2.2: Illustration of the bottom-up dynamic programming algorithm EvalDP, for a PTIME probability computation over the simple tree pattern query Q:`/A//B`

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 [11, 16], but is not applicable to queries or data that involve correlations.

2.2.2 DNF Probability Estimation

In [35], 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 [35] 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 consider 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 biased approximation in [52], 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 [35] that is obtained from Hoeffding's inequality requires a running time growing in $O(n^3 \ln n)$ in the number of matches (i.e., in the number of clauses of the lineage formula).

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 [32] (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 [39], and as deployed in ProApproX.

2.2.3 Systems

At the practical level, a full-fledged probabilistic semistructured data management system is still missing. Hollander and van Keulen [27] 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 [35].

Initial ideas leading to this work were presented in [52,54] though the specificity of our approach, using a cost estimator to optimize the running time of the (approximate) probability computing, is fully novel.

Chapter 3

Algorithms and Cost Models

In this chapter 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. A value for c_{alg} is measured experimentally by varying the size of synthetic formulas and recording the processing time of the actual implementation of the algorithm on a given machine (see Section 5.1.1 for further details on obtaining algorithms constants). We only consider sequential processing here, see the conclusion for discussions about parallelization. We give in Table 3.1 a summary of the notation used and in Table 3.2 a summary of the cost models of all algorithms.

3.1 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.

3.1.1 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 (Table 3.3). The query Q1 of Example 1 in Section B.2.1, the related lineage formula $\varphi_1 = \varphi_1 = c_1 \vee c_2 \vee c_3 = (e_5 \wedge e_1 \wedge e_2) \vee (e_5 \wedge e_2 \wedge e_3) \vee (e_5 \wedge e_1 \wedge e_4)$

Table 3.1: 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
s	an assignment of truth values to event variables (literals) of 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/reliability factor
ℓ	lower bound on φ's probability
$\tilde{p}$	an approximation of the probability of φ
cost_{alg}	cost of an algorithm alg
c_{alg}	cost constant for algorithm alg

Table 3.2: Cost models and cost constants

Algorithm alg	Type of evaluation	cost_{alg}	c_{alg} (ms)
Naïve	exact	$c_{\text{naïve}} \times 2^N \times L$	$4 \cdot 10^{-5}$
Sieve	exact	$c_{\text{sieve}} \times 2^m \times \frac{L}{m}$	$5 \cdot 10^{-5}$
Naïve Monte-Carlo	additive approximation	$c_{\text{addMD}} \times \ln \frac{2}{\delta} \times \frac{L}{\varepsilon^2}$	$4 \cdot 10^{-5}$
Biased Monte-Carlo	multiplicative approximation	$c_{biasedMC} \times \ln \frac{2}{\delta} \times \frac{L}{\ell^2 \varepsilon^2}$	$4 \cdot 10^{-5}$
Self-Adjusting Coverage	multiplicative approximation	$c_{\text{coverage}} \times \ln \frac{2}{\delta} \times \frac{(1+\varepsilon) \times L}{\varepsilon^2}$	10^{-3}

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 (Table 3.3).

Possible World						Probability
e_1	e_2	e_3	e_4	e_5	φ_1	
true	true	false	false	true	true	0.0162
true	true	true	false	true	true	0.0162
true	true	true	true	true	true	0.0378
false	true	true	false	true	true	0.0648
...						

Table 3.3: Some satisfying truth value assignments (worlds) for $\varphi_1 = c_1 \vee c_2 \vee c_3 = (e_5 \wedge e_1 \wedge e_2) \vee (e_5 \wedge e_2 \wedge e_3) \vee (e_5 \wedge e_1 \wedge e_4)$

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 $4 \cdot 10^{-5}$ ms (see Table 3.2 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 $c_{\text{naïve}} \times 2^{10} \times 500 \approx 20$ 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 adapted.

3.1.2 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:

$$cost = c_{\text{sieve}} \times 2^m \times \frac{L}{m}$$

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 [25].

3.2 Approximation Algorithms

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

$$p - \varepsilon \leqslant \tilde{p} \leqslant p + \varepsilon$$

Similarly, for fixed ε, δ, $\tilde{p}$ is a multiplicative ε-approximation of p with probabilistic guarantee $1-\delta$ if, with probability at least $1-\delta$:

$$(1-\varepsilon) \times p \leqslant \tilde{p} \leqslant (1+\varepsilon) \times p$$

In Section 4.2, we explain how to turn additive guarantees into multiplicative ones.

3.2.1 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 (Algorithm 1). 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 (that forms a special case of one of the Bernstein's inequality) [26], we have:

$$\Pr\left(|\tilde{p} - p| > \varepsilon\right) \leqslant 2\mathrm{e}^{-2t\varepsilon^2}$$

$\Pr\left(|\tilde{p} - p| > \varepsilon\right)$ is the reliability factor δ. 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:

$$\mathrm{cost}_{\mathrm{addMD}} = c_{\mathrm{addMD}} \times \ln \frac{2}{\delta} \times \frac{L}{\varepsilon^2}$$

Algorithm 1 Additive Monte-Carlo

$trials \leftarrow (\ln 2 - \ln \delta)/(2\varepsilon^2)$
$t \leftarrow 0$
$success \leftarrow 0$
while $t \leqslant trials$ **do**
  randomly pick an assignment s_i
  if s_i satisfies the DNF **then**
    $success \leftarrow success + 1$
  end if
  $t \leftarrow t + 1$
end while
return $\tilde{p} = \frac{success}{t}$

In ProApproX, the algorithm is optimized such that it does not draw a complete assignment at once. The program goes through the clauses and picks assignment just for the current clause, verifies if its evaluation is true (which implies that the DNF is satisfied and renders a stop condition for the current trial), then records a success in such a case. Otherwise, and being always in the same trial, the assignment for the next clause is picked (Algorithm 2).

Algorithm 2 Additive Monte-Carlo - Optimized

1: $trials \leftarrow (\ln 2 - \ln \delta)/(2\varepsilon^2)$
2: $t \leftarrow 0$
3: $success \leftarrow 0$
4: **while** $t \leqslant trials$ **do**
5: $i \leftarrow 1$
6: trial:
7: **while** $i \leqslant m$ (m: number of clauses in the DNF) **do**
8: draw assignment for each literal in current clause c_i (if the literal is not yet assigned in the current trial)
9: **if** c_i is true **then**
10: $success \leftarrow success + 1$
11: break trial.
12: **end if**
13: **end while**
14: $t \leftarrow t + 1$
15: **end while**
16: **return** $\tilde{p} = \frac{success}{t}$

Despite the fact that this algorithm is linear in the size of the DNF, it typically becomes costly or leads to bad accuracy for cases when the probability is low. This typical hard case for the naive Monte Carlo happens when the number of satisfying assignments is exponentially small with respect to the total (huge) number of possible assignments. From an approximation point of view, we would need exponentially many samples from the set of possible assignments to get an accurate number of successes. In such a case, a biased approximation is more likely to lead to better results with a noticeably lower cost.

3.2.2 Multiplicative/Biased Monte-Carlo

The probability of a DNF could be estimated by running a biased multiplicative approximation [35] (with m the number of clauses for the DNF):

$$\sum_{i=1}^{m} \Pr(c_i) \times \Pr\left(\bigwedge_{j=1}^{i-1} \neg(c_j) \mid c_i\right)$$

The first term is easy to compute by just gathering the probabilities of all events involved in the clause; the second term can be approximated by conducting biased draws to consider the probability that none of the preceding clauses is satisfied, given that the literals of the current clause c_i are set to a valuation that satisfies c_i.

Similarly, the Hoeffding's inequality for an approximation within a multiplicative interval [35] would be:

$$Pr(\tilde{p} \in [(1-\varepsilon)p; (1+\varepsilon)p]) \leqslant 2m \exp^{m^2 - 2t\varepsilon^2}$$

Given $\delta = Pr(\tilde{p} \in [(1-\varepsilon)p; (1+\varepsilon)p])$, the number of trials t needed to perform an (ε, δ) multiplicative approximation is:

$$t \leqslant m^2 \frac{\ln 2m - \ln \delta}{2\varepsilon^2}$$

Note that for this algorithm as for the previous one, based on Hoeffding's inequalities, we can compute the error reached at a given trial:

$$\varepsilon \leqslant m \sqrt{\frac{\ln 2m - \ln \delta}{2t_{current}}}$$

In an early version of ProApproX [52], where we proposed more options to define the stage where the approximation should stop and render the result, we based the decision on the error reached in the current trial. For example, a user could define a convergence indicator that makes an approximation stop when say 100 simultaneous trials fall into a given error margin.

Recall the example DNF $\varphi_1 = c_1 \vee c_2 \vee c_3 = (e_5 \wedge e_1 \wedge e_2) \vee (e_5 \wedge e_2 \wedge e_3) \vee (e_5 \wedge e_1 \wedge e_4)$. We illustrate the biased Monte-Carlo presented in [35] on the example:

$$Pr(F) = Pr(c_1) + Pr(c_2)Pr(\neg c_1 | c_2) + Pr(c_3)Pr(\neg c_1 \wedge \neg c_2 | c_3)$$

The parts that might cost a significant time are the terms to approximate, in other words: $Pr(\neg c_1|c_2)$ and $Pr(\neg c_1 \wedge \neg c_2|c_3)$. From c_2 to c_m, each of these terms evaluates the chance that when the current clause is true (i.e., for all possible worlds/truth assignments where c_i is true), none of the previous clauses is also true. This probability is evaluated by assigning truth values for the literals of c_i so that the latter is true, and drawing random assignments for the previous clauses one by one (Algorithm 3). Again, this evaluation is performed for each of the remaining clauses, which explains the factor m in the cost model for the biased Monte-Carlo:

$$cost_{biasedMC} = c_{biasedMC} . \frac{L}{m} . m^2(m) \frac{\ln(\frac{2m}{\delta})}{\varepsilon^2}$$

The Biased Monte-Carlo was included in the early versions of ProApproX, then replaced by a much more efficient multiplicative approximation, the *Self-Adjusting Coverage Algorithm*, that we describe in the upcoming section.

Algorithm 3 Biased Monte-Carlo

1: U_j is the set of assignments that satisfy the clause c_j
2: $\tilde{p} \leftarrow Pr(c_1)$
3: $trials \leftarrow m^2(\ln 2m - \ln \delta)/(2\varepsilon^2)$
4: **run:**
5: **for all** $c_i \in (c_2, ..., c_m)$ **do**
6: assign the literals of c_i so that c_i is true
7: $t \leftarrow 1$
8: **trial:**
9: **while** $(t \leqslant trials)$ **do**
10: $j \leftarrow 1$
11: $success \leftarrow 0$
12: **step:**
13: **while** $(j < i)$ **do**
14: pick a random assignment s for c_j, so that it extends the existing one (literals to make c_i true) and also all the assignments chosen for a c_k with $k < j$
15: **if not** $s \in U_j$ **then**
16: $j \leftarrow j + 1$
17: **else**
18: break step
19: **end if**
20: **end while**
21: **if** $j = i$ **then**
22: $success \leftarrow success + 1$
23: **end if**
24: $t \leftarrow t + 1$
25: **end while**
26: $\tilde{p} \leftarrow \tilde{p} + Pr(c_i) \times (success/trials)$
27: **end for**
28: **return** $\tilde{p}$

3.2.3 Self-Adjusting Coverage Algorithm

This algorithm was introduced by Karp, Luby, and Madras [32] and is a fully-polynomial randomized approximation scheme (FPRAS), which means it produces a multiplicative estimate of the probability. The Self Adjusting Coverage, described in Algorithm 4, uses a time estimator, to delimit the number of biased trials, so that with a high probability we get into a good sampling. These biased trials are designed to weight the quantity of dependency (intersection) between the different clauses and take it into consideration while computing their union (probability of

the DNF). The algorithm processes the clauses randomly and stops the processing after reaching a pre-computed number of total trials t:

- Self-Adjusting Coverage Algorithm Theorem I [32]. *When* $\varepsilon < 1$ *and* $t = (8.(1+\varepsilon).m\ln(3/\delta))/(1-\varepsilon^2/8)\varepsilon^2$, *the self-adjusting coverage algorithm is an* (ε, δ) *approximation algorithm when estimator* $\tilde{p}$ *is used.*

- Self-Adjusting Coverage Algorithm Theorem II [32]. *When* $\varepsilon < 1$ *and* $t = (8.(1+\varepsilon).m\ln(2/\delta))/\varepsilon^2$, *the self-adjusting coverage algorithm is an* (ε, δ) *approximation algorithm when estimator* $\tilde{p'}$ *is used.*

Since each trial requires evaluating one clause of φ, we have:

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

As can be seen on Table 3.2, 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, 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.

Algorithm 4 Self-Adjusting Coverage Algorithm adapted for the DNF probability problem [31]

$\tilde{p} \leftarrow 0$
$\tilde{Pr'}(\varphi) \leftarrow 0$
$gtime \leftarrow 0$ ($gtime$ counts the global number of steps executed)
$Total \leftarrow 0$
$t' \leftarrow 0$
(t is set as specified in Theorem I or Theorem II)
loop {**trial:**}
  randomly choose (c, s): a clause c following the (aligned) distribution of clauses and an assignment s that satisfies c and following the distribution of variables
  $t \leftarrow 0$
  loop {**step:**}
    $t \leftarrow t + 1$
    $gtime \leftarrow gtime + 1$
    if $gtime > t$ **then**
      go to finish
    end if
    randomly choose $c_j \in c_1, ..., c_m$ with probability $1/m$
    if not s satisfies c_j **then**
      go to step
    end if
    TOTAL$\leftarrow gtime$
    $t' \leftarrow t' + 1$
    $f(s) \leftarrow t/m$
    $\tilde{p} \leftarrow \tilde{p} + |U|.f(s)$ `//with` $|U| = \sum_{i=1}^{m} Pr(c_i)$
    go to trial
  end loop
end loop
finish:
return $\tilde{p}$
(Equivalently $\tilde{p} \leftarrow (TOTAL.|U|)/(m.t')$)
return $\tilde{p'} \leftarrow (t.|U|)/(m.t')$

Chapter 4

Computation Strategies

4.1 Compiling The Probabilistic Lineage

We explain in this section how ProApproX decomposes a DNF formula φ into a compilation tree formed of simpler subformulas. We assume in this section that no contradiction appears in φ (a clause that contains both x and $\neg x$ and is removed from φ). 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.*

4.1.1 Analyzing and Decomposing the DNF

Our compilation 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 formulas in DNF and whose internal nodes are Boolean operations (independent disjunction $\textcircled{\vee}$, independent conjunction $\textcircled{\wedge}$, and mutually exclusive disjunction $\oplus$) 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

$$\varphi_2 = ((e_1 \wedge e_2) \vee (e_1 \wedge e_3)) \textcircled{\vee} ((\neg e_4 \wedge e_5) \vee (e_4 \wedge e_6) \vee (e_4 \wedge e_7))$$

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 $\varphi' = c \barwedge (c'_1 \vee \cdots \vee c'_m)$ where $\barwedge$ denotes independent conjunction: c is independent of every c_i. Note that both operands of $\barwedge$ 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

$$\varphi_2 = (e_1 \barwedge (e_2 \vee e_3)) \barvee ((\neg e_4 \wedge e_5) \vee (e_4 \wedge e_6) \vee (e_4 \wedge e_7))$$

Inconsistency

Our final operator looks for a partition of a DNF formula into two inconsistent subformulas, i.e., a rewriting of φ into $\varphi' \oplus \varphi''$ 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

$$\varphi_2 = (e_1 \barwedge (e_2 \vee e_3)) \barvee ((\neg e_4 \wedge e_5) \oplus ((e_4 \wedge e_6) \vee (e_4 \wedge e_7)))$$

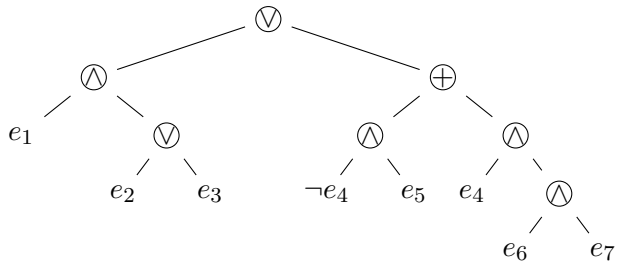
Algorithm 5 ApplyOperators()

```
1: INPUT: a compilation tree CTree
2: treeSize ← 0
3: while treeSize < CTree.size() do
4:    treeSize ← CTree.size()
5:    CTree ← partition(CTree)
6:    CTree ← factorize(CTree)
7:    CTree ← inconsistent(CTree)
8: end while
9: return CTree
```

ProApproX repeatedly applies these three operations until no further rewritings are possible (Algorithm 5). We obtain thus for φ_2:

$$\varphi_2 = (e_1 \barwedge (e_2 \barvee e_3)) \barvee ((\neg e_4 \barwedge e_5) \oplus (e_4 \barwedge (e_6 \barvee e_7)))$$

This defines a compilation tree for φ_2 in a straightforward manner; inner nodes are the Boolean operations $\barvee$, $\barwedge$, $\oplus$ (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 and do not make any claim at optimality of the simplification obtained.

These compilation 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:

$$\Pr(\psi_1 \bigvee \psi_2) = 1 - (1 - \Pr(\psi_1)) \times (1 - \Pr(\psi_2))$$
$$\Pr(\psi_1 \bigwedge \psi_2) = \Pr(\psi_1) \times \Pr(\psi_2)$$
$$\Pr(\psi_1 \bigoplus \psi_2) = \Pr(\psi_1) + \Pr(\psi_2)$$

4.1.2 From a Compilation Tree to an Evaluation Tree

In general, the compilation 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 compilation trees into evaluation trees 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:

- 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).
- It is sometimes not worth 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 4.3)

.

4.2 Propagation of Approximation Parameters

Our overall objective is to obtain an exact or, when hard, an approximate value of the probability of a formula φ in DNF, given an approximation tolerance ε

and a 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 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 what follows, we establish the decision functions to propagate the error and confidence between two nodes linked by a given logic operator. To perform the propagation through the whole evaluation tree, the decision is then done recursively i.e., the structure is viewed as a binary decision tree.

We establish the different conditions for correct error allocation for children of $\otimes$, $\oslash$, or $\oplus$ nodes, in the case when:

- both nodes are approximated;
- one node is approximated and the other is evaluated with an exact computation.

We also explain how to turn an additive approximation into a multiplicative one and vice-versa, and how the approximation guarantee $1 - \delta$ is propagated.

4.2.1 Propagation Between Approximated Nodes

In this section, we state the propagation condition then the proof based on which the decision strategy is performed in ProApproX to allocate error values for children of $\otimes$, $\oslash$, or $\oplus$ nodes, when:

- both use an additive approximation;
- both use a multiplicative approximation;

The last case, when one node is additively approximated and the other uses a multiplicative approximation, is analyzed in the next section, where we practically use a trick to make the multiplicative approximation produce an additive precision and vice-versa.

Both Nodes Evaluated With Additive Approximation

Proposition 1. *Let $\varphi = \psi_1 \otimes \psi_2$, and assume $\tilde{p}_1$ and $\tilde{p}_2$ are additive 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 an additive approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2 \leqslant \varepsilon$.*

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 additive approximations of p_1 and p_2, and

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

Then:

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

And thus:

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

Lower Bound.

$LB = [1 - [((1-p_1)+\varepsilon_1)((1-p_2)+\varepsilon_2)]$
$= [1 - [(1-p_1)(1-p_2) + (1-p_1)\varepsilon_2 + (1-p_2)\varepsilon_1 + \varepsilon_1\varepsilon_2]$
$= \underbrace{1-(1-p_1)(1-p_2)}_{=\mathrm{p}} - ((1-p_1)\varepsilon_2 + (1-p_2)\varepsilon_1 + \varepsilon_1\varepsilon_2)$

Let $\lambda = ((1-p_1)\varepsilon_2 + (1-p_2)\varepsilon_1 + \varepsilon_1\varepsilon_2)$

When p_1 and p_2 approach zero, λ approaches its maximum value:

$$LB = p - (\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2)$$

Therefore, to have $LB \geqslant \tilde{p}$, we need to set ε_1 and ε_2 such that $(\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2) \leqslant \varepsilon$

Upper Bound.

$UB = [1 - [((1-p_1)-\varepsilon_1)((1-p_2)-\varepsilon_2)]$
$= [1 - [(1-p_1)(1-p_2) - (1-p_1)\varepsilon_2 - (1-p_2)\varepsilon_1 + \varepsilon_1\varepsilon_2]$
$= \underbrace{1-(1-p_1)(1-p_2)}_{=\mathrm{p}} + ((1-p_1)\varepsilon_2 + (1-p_2)\varepsilon_1 - \varepsilon_1\varepsilon_2)$

Let $\lambda = ((1-p_1)\varepsilon_2 + (1-p_2)\varepsilon_1 - \varepsilon_1\varepsilon_2)$

Here also,when p_1 and p_2 approach zero, λ approaches its maximum value:

$$UB = p + (\varepsilon_1 + \varepsilon_2 - \varepsilon_1\varepsilon_2)$$

Therefore $UB \leqslant \tilde{p}$ whenever $\varepsilon_1 + \varepsilon_2 - \varepsilon_1\varepsilon_2 \leqslant \varepsilon$

Conclusion: Values for ε_1 and ε_2 are chosen under the following general condition:

$$\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2 \leqslant \varepsilon$$

□

Proposition 2. *Let $\varphi = \psi_1 \oplus \psi_2$, and assume $\tilde{p}_1$ and $\tilde{p}_2$ are additive 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 an additive approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon_1 + \varepsilon_2 = \varepsilon$.*

Proof.

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

Applying the $\oplus$ operator we have:

$$p_1 + p_2 - \varepsilon_1 - \varepsilon_2 \leqslant \tilde{p_1} + \tilde{p_2} \leqslant p_1 + p_2 + \varepsilon_1 + \varepsilon 2$$

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

The propagation condition in this case is quite straightforward:

$$\varepsilon_1 + \varepsilon_2 = \varepsilon$$

□

For the $\owedge$ 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 the case:

Proposition 3. *Let $\varphi = \psi_1 \owedge \psi_2$, and assume $\tilde{p}_1$ is an additive approximation of $\Pr(\psi_1)$ to a factor of ε_1. Then $\tilde{p}$ is an additive approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon = \varepsilon_1$.*

Both Nodes Evaluated With Multiplicative Approximation

Proposition 4. *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:

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

Lower Bound.
Let α be $1-[(1-(1-\varepsilon_1)p_1)(1-(1-\varepsilon_2)p_2] = 1-[[1-p_1+\varepsilon_1 p_1][1-p_2+\varepsilon_2 p_2]]$
$= 1-[[(1-p_1)+\varepsilon_1 p_1][(1-p_2)+\varepsilon_2 p_2]]$
$= 1-[(1-p_1)(1-p_2)+(1-p_1)\varepsilon_2 p_2+(1-p_2)\varepsilon_1 p_1+\varepsilon_1\varepsilon_2 p_1 p_2]$
$= 1-(1-p_1)(1-p_2)-(1-p_1)\varepsilon_2 p_2-(1-p_2)\varepsilon_1 p_1-\varepsilon_1\varepsilon_2 p_1 p_2$
We have:
$\alpha = \underbrace{1-(1-p_1)(1-p_2)}_{=\mathrm{p}} -[(1-p_1)\varepsilon_2 p_2+(1-p_2)\varepsilon_1 p_1+\varepsilon_1\varepsilon_2 p_1 p_2]$

Let $\lambda = [(1-p_1)\varepsilon_2 p_2+(1-p_2)\varepsilon_1 p_1+\varepsilon_1\varepsilon_2 p_1 p_2]$

Given $\frac{\lambda}{p} = \frac{(1-p_1)\varepsilon_2 p_2+(1-p_2)\varepsilon_1 p_1+\varepsilon_1\varepsilon_2 p_1 p_2}{p_1+p_2-p_1 p_2}$; let $\Pi = \max(p_1, p_2)$; we have $p \geqslant \Pi$.

We write λ as $\varepsilon_1[\varepsilon_2 p_1 p_2+(1-p_2)p_1]+\varepsilon_2 p_2(1-p1) = \varepsilon_1 p_1[1+p_2(\varepsilon_2-1)]+\varepsilon_2 p_2(1-p_1)$
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 $\lambda \leqslant \varepsilon p$.

Upper Bound.
We proceed similarly for the upper bound. Let α be $1-[(1-(1+\varepsilon_1)p_1)(1-(1+\varepsilon_2)p_2)$

$= 1-[[1-p_1-\varepsilon_1 p_1][1-p_2-\varepsilon_2 p_2]]$
$= 1-[[(1-p_1)-\varepsilon_1 p_1][(1-p_2)-\varepsilon_2 p_2]]$
$= 1-[(1-p_1)(1-p_2)-(1-p_1)\varepsilon_2 p_2-(1-p_2)\varepsilon_1 p_1+\varepsilon_1\varepsilon_2 p_1 p_2]$
$= 1-(1-p_1)(1-p_2)+(1-p_1)\varepsilon_2 p_2+(1-p_2)\varepsilon_1 p_1-\varepsilon_1\varepsilon_2 p_1 p_2$

We have:
$\alpha = \underbrace{1-(1-p_1)(1-p_2)}_{=\mathrm{p}} + [(1-p_1)\varepsilon_2 p_2 + (1-p_2)\varepsilon_1 p_1 - \varepsilon_1\varepsilon_2 p_1 p_2]$

Let $\lambda = [(1-p_1)\varepsilon_2 p_2 + (1-p_2)\varepsilon_1 p_1 - \varepsilon_1\varepsilon_2 p_1 p_2]$ and by a similar reasoning we want to find a condition under which we can attribute values to ε_1 and ε_2:

$\frac{\lambda}{p} = \frac{(1-p_1)\varepsilon_2 p_2 + (1-p_2)\varepsilon_1 p_1 - \varepsilon_1\varepsilon_2 p_1 p_2}{p}$; and again, let $\Pi = \max(p_1, p_2)$ so obviously $p \geqslant \Pi$.

We write λ as $= \varepsilon_1 p_1[1 - p_2(1-\varepsilon_2)] + \varepsilon_2 p_2(1-p_1)$
obviously: $\lambda \leqslant \varepsilon_1\Pi + \varepsilon_2\Pi$; $\frac{\lambda}{p} \leqslant \varepsilon_1 + \varepsilon_2\frac{p}{p} \leqslant \varepsilon_1 + \varepsilon_2 = \varepsilon$ and $\lambda \leqslant \varepsilon p$. Therefore $\alpha \leqslant p(1+\varepsilon)$ holds whenever:

$$\varepsilon_1 + \varepsilon_2 \leqslant \varepsilon$$

□

In practice, when we propagate an ε factor down the tree, we look for different combinations of ε_1 and ε_2 that sum to 1, 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 following 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 5. *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. Starting from:

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

we get to define bounds for the approximated probability $\tilde{p}$ as follows:

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

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 $\varepsilon_2 = \varepsilon$. □

Again, we state the propagation condition for the $\barwedge$ operator linking a formula in DNF and a conjunction:

Proposition 6. *Let $\varphi = \psi_1 \barwedge \psi_2$, 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 $\varepsilon = \varepsilon_1$.*

4.2.2 Propagation Between Exact and Approximated Nodes

Proposition 7. *Let $\varphi = \psi_1 \textcircled{\vee} \psi_2$, and assume $\tilde{p}_1$ is an additive approximation of $\Pr(\psi_1)$ to a factor of ε_1, and that $\Pr(\psi_2)$,noted p_2, is computed exactly. Then $1-(1-\tilde{p}_1)(1-p_2)$ is an additive approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon_1 = \varepsilon$.*

Proof. Starting from: $1 - p_1 - \varepsilon_1 \leqslant (1 - \tilde{p_1}) \leqslant 1 - p_1 + \varepsilon_1$, and knowing that $(1 - p_2) \geqslant 0$ then:

$(1 - p_1 - \varepsilon_1)(1 - p_2) \leqslant (1 - \tilde{p}_1)(1 - p_2) \leqslant (1 - p_1 + \varepsilon_1)(1 - p_2)$

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

$\underbrace{1 - (1 - p_1)(1 - p_2)}_{=\mathrm{p}} - \underbrace{\varepsilon_1(1 - p_2)}_{\leqslant \varepsilon} \leqslant 1 - (1 - \tilde{p_1})(1 - p_2) \leqslant \underbrace{1 - (1 - p_1)(1 - p_2)}_{= p} +$
$\underbrace{\varepsilon_1(1 - p_2)}_{\leqslant \varepsilon}$

To guarantee an additive error we must have $\varepsilon_1(1 - p_2) \leqslant \varepsilon$.
If we decide to allocate a value to ε_1 that could reach at most the value of ε it is then always true that $\varepsilon_1(1 - p_2) \leqslant \varepsilon$. Without any a-priori information about the quantity p_2, the best allocation that we maintain is then:

$$\varepsilon_1 = \varepsilon$$

□

Proposition 8. *Let $\varphi = \psi_1 \bigoplus \psi_2$, $\tilde{p}_1$ is an additive approximation of $\Pr(\psi_1)$ to a factor of ε_1, and p_2 is an exact value for $\Pr(\psi_2)$. Then $\tilde{p}_1 + p_2$ is an additive approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon_1 = \varepsilon$.*

Proof. $\tilde{p}_1$ is bounded by:$p_1 - \varepsilon_1 \leqslant \tilde{p_1} \leqslant p_1 + \varepsilon_1$

$\Leftrightarrow \underbrace{p_1 + p_2}_{=\mathrm{p}} - \underbrace{\varepsilon_1}_{=\varepsilon} \leqslant \tilde{p_1} + p_2 \leqslant \underbrace{p_1 + p_2}_{= p} + \underbrace{\varepsilon_1}_{=\varepsilon}$

□

Proposition 9. *Let $\varphi = \psi_1 \textcircled{\vee} \psi_2$, and assume $\tilde{p}_1$ is a multiplicative approximation of $\Pr(\psi_1)$ to a factor of ε_1, and p_2 is an exact value for $\Pr(\psi_2)$. Then $1 - (1 - \tilde{p}_1)(1 - p_2)$ is a multiplicative approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon = \varepsilon_1$.*

Proof. Given that: $p_1 - \varepsilon_1 p_1 \leqslant \tilde{p_1} \leqslant p_1 + \varepsilon_1 p_1$

then $1 - p_1 - \varepsilon_1 p_1 \leqslant (1 - \tilde{p_1}) \leqslant 1 - p_1 + \varepsilon_1 p_1$

and also $(1 - p_1 - \varepsilon_1 p_1)(1 - p_2) \leqslant (1 - \tilde{p_1})(1 - p_2) \leqslant (1 - p_1 + \varepsilon_1 p_1)(1 - p_2)$

getting to bounds for $\tilde{p}$:

$1 - ((1d - p_1) + \varepsilon_1 p_1)(1 - p_2) \leqslant 1 - (1 - \tilde{p_1})(1 - p_2) \leqslant 1 - ((1 - p_1) - \varepsilon_1 p_1)(1 - p_2)$

$$\iff$$

$$\underbrace{1-(1-p_1)(1-p_2)}_{=\mathrm{p}} - \underbrace{\varepsilon_1 p_1(1-p_2)}_{\leqslant \varepsilon p} \leqslant 1-(1-\tilde{p_1})(1-p_2) \leqslant \underbrace{1-(1-p_1)(1-p_2)}_{=\mathrm{p}} + \underbrace{\varepsilon_1 p_1(1-p_2)}_{\leqslant \varepsilon p}$$

Here, we would like to find the condition for ε_1 under which the relation $\varepsilon_1 p_1(1-p_2) \leqslant \varepsilon p$ holds. For this we just need to verify that: $\frac{\varepsilon_1 p_1(1-p_2)}{p} \leqslant \varepsilon$
Let $\Pi = \max(p_1, p_2)$, then we know that: $p \geqslant \Pi$
$p_1(1-p_2) \leqslant \Pi$ because $p_1 \leqslant \Pi$ and $(1-p_2) \leqslant 1$, also $\varepsilon_1 \leqslant 1$ therefore: $\varepsilon_1 p_1(1-p_2) \leqslant \Pi$

The following then holds: $\frac{\varepsilon_1 \overbrace{p_1(1-p_2)}^{\leqslant \Pi}}{\underbrace{p}_{\geqslant \Pi}} \leqslant \varepsilon$ providing the needed condition that $\varepsilon_1 \leqslant \varepsilon$

□

Proposition 10. *Let $\varphi = \psi_1 \bigoplus \psi_2$, $\tilde{p_1}$ is a multiplicative approximation of $\Pr(\psi_1)$ to a factor of ε_1, and p_2 is an exact value for $\Pr(\psi_2)$. Then $\tilde{p_1}+p_2$ is a multiplicative approximation of $\Pr(\varphi)$ to a factor of ε if $\varepsilon = \varepsilon_1$.*

Proof. $p_1 - \varepsilon_1 p_1 \leqslant \tilde{p_1} \leqslant p_1 + \varepsilon_1 p_1$
$\underbrace{p_1+p_2}_{\mathrm{p}} - \underbrace{\varepsilon_1 p_1}_{\leqslant \varepsilon p} \leqslant \tilde{p_1} + p_2 \leqslant \underbrace{p_1+p_2}_{\mathrm{p}} + \underbrace{\varepsilon_1 p_1}_{\leqslant \varepsilon p}$
$\frac{\varepsilon_1 p_1}{p} \leqslant \varepsilon p$ holds because $p = p_1 + p_2 - p_1 p_2 \geqslant p_1$. Thus for this propagation, we simply have to optimally choose: $\varepsilon_1 = \varepsilon$ as $p_1 \leqslant 1$.

□

4.2.3 Combining Additive and Multiplicative Approximations

In such a situation, we are going to simply switch an additive approximation for a multiplicative one and vice versa. Thereby, we will not need to set new propagation conditions, but rather cast the approximation so that both nodes are perceived to use the same type of algorithm. This observation was achieved by a first interest in examining the relevance of using additive algorithms to achieve a multiplicative error or the other way around.

It is possible to run a multiplicative approximation to ensure additive error at the root by simply using the same tolerance given by the user and share it between the nodes following the propagation conditions that we are exposing in this section. This is because the error bound produced by a multiplicative approximation that uses the error ε will always be within the additive bound produced by the same ε:

Bounds of an additive approximation: $p \pm \varepsilon$
Bounds of a multiplicative approximation: $p \pm \varepsilon p$
p is a value $\leqslant 1$ therefore, it is always true that: $\varepsilon p \leqslant \varepsilon$

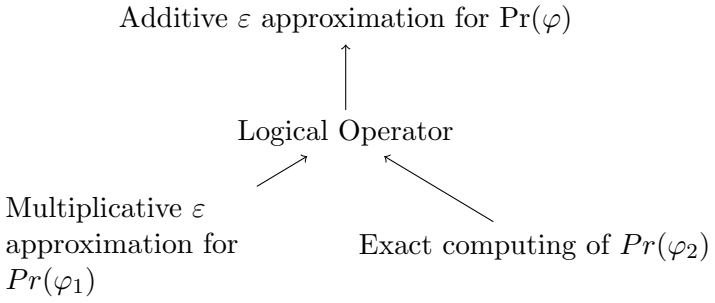


Figure 4.1: Producing the additive error via multiplicative approximations

In other cases, resorting to an execution plan using additive algorithms with the ultimate goal of producing a multiplicative tolerance, might 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 in this case, 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

$$\text{cost}_{\text{MulMC}} = C_{\text{addMD}} \times \ln \frac{2}{\delta} \times \frac{L}{\ell^2 \varepsilon^2}.$$

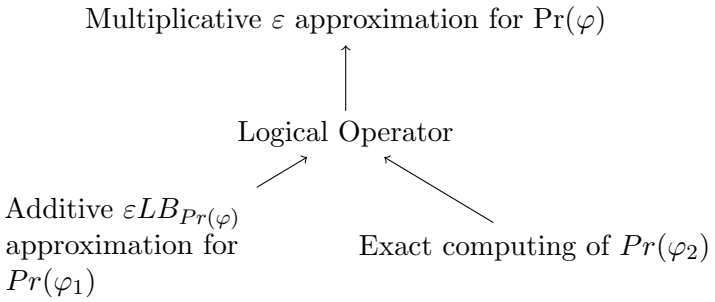


Figure 4.2: Producing the multiplicative error via additive approximations

Generally speaking, if we want to combine multiplicative and additive approximations at the nodes level, we can follow the same reasoning and use a unary operator (εLB_p) to convert promptly a pre-allocated multiplicative error into a sufficient additive tolerance (The unary operator is ε for the opposite case: *"from a multiplicative tolerance to an additive tolerance"*).

4.2.4 Propagation of Error Bounds: Conclusions

	Additive for both nodes	Multiplicative for both nodes
Independent OR ($\veebar$)	$\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$	$\varepsilon_1 + \varepsilon_2$
Exclusive OR ($\oplus$)	$\varepsilon_1 + \varepsilon_2$	$\max(\varepsilon_1, \varepsilon_2)$

Additive→Multiplicative	Multiplicative→Additive
εLB_p	ε

4.2.5 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 $\delta_1 = \delta_2 = 1 - \sqrt{1 - \delta}$.

4.3 Exploring the Space of Possible Evaluation Plans

We now have all the 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 4.1, with the same root, where every leaf is associated with an algorithm among the four described in Chapter 3, 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 4.2. 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 cost 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 4.1.)

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.

4.3.1 A Randomized Approach

On a fixed subtree, we first allocate ε and δ's equitably following the constraints of Section 4.2. Being under the propagation condition, many possible values for ε_1 and ε_2 can be found. We can find the best assignment locally for the direct descendent of a node through 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 would tend to infinity, the randomized approach for finding a "good" evaluation plan will definitely be the way to follow.

Inspired by query optimization techniques, we choose the approach of exploring the space of evaluation plans by sampling a large number of them, evaluating the cost of each of them, and choose the best one found, which is thus not necessarily optimal, but good enough in practice. We observed in the experiments that this process thus leads to better results than always choosing, say, the full evaluation tree.

Our exploration of the search space is based on the following principles:

1. We always consider the tree formed of a single root node and the full compilation tree (Section 4.1.2) during the search.

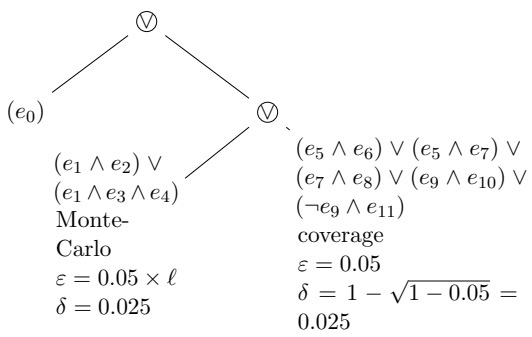


Figure 4.3: Example $(0.1, 0.05)$-execution plan 1 for φ_3

2. In addition, we generate a sample of $1,000$ (or less, see further in point 5) subtrees obtained by deciding, uniformly at random over all nodes of the evaluation tree, either to expand an internal node into its children, or to keep it as is and evaluate the formula at this level.

3. As a rule, we never expand clauses when they are on their own, since it is already easy to compute the probability of a clause in isolation.

4. Once a node is assigned with an error and confidence quantities (at random but under the constraints of Section 4.2), we choose the best algorithm for its subformula. If this algorithm is an exact algorithm, we reallocate the unused ε's and δ's to other branches of the tree.

5. So as not to spend too much time in the search, we end the search if we have exceeded one tenth of the time given by the cost of the best estimated plan found so far.

Example 3. *Consider the formula* $\varphi_3 = e_0 \vee (e_1 \wedge e_2) \vee (e_1 \wedge e_3 \wedge e_4) \vee (e_5 \wedge e_6) \vee (e_5 \wedge e_7) \vee (e_7 \wedge e_8) \vee (e_9 \wedge e_{10}) \vee (\neg e_9 \vee e_{11})$. *We present in Figures 4.3 and 4.4 possible evaluation plans for* φ_3*, given multiplicative error bounds* $\varepsilon = 0.1$ *and probabilistic guarantee* $1 - \delta = 0.95$. *Exact computations of individual conjunctions are not shown. This is for illustration only; on subformulas of such small size, the optimal strategy would be to expand the tree as much as possible, and then use one of the exact algorithms, based on the cost model.*

4.3.2 The Deterministic Case

In case the propagation of approximation parameters is decided statically under the constraints of Section 4.2, it becomes easy to hit the best evaluation plan. The exploration of the space of possible evaluation plans is implemented under the deterministic case in ProApproX.

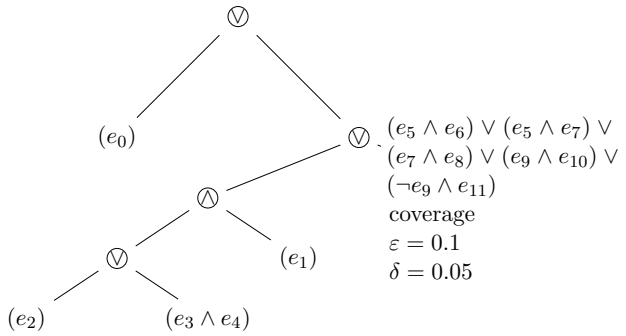


Figure 4.4: Example $(0.1, 0.05)$-execution plan 2 for φ_3

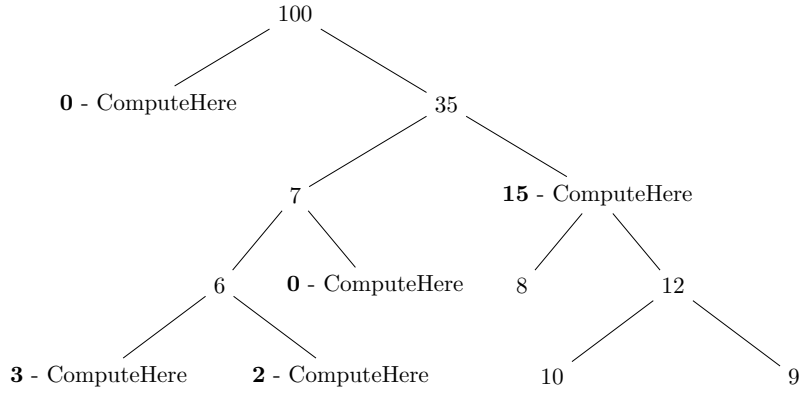


Figure 4.5: An example best evaluation plan in a deterministic case.

Step 1: Propagation of approximation values and allocation of algorithms Our approach begins by propagating values to every single node of the compilation tree (i.e., including inner nodes), before deciding on algorithms. At a given node, when a DNF has more than two clauses, it will receive a propagated portion of ε and δ following the logical operator of the parent node. We do not propagate approximation parameters to nodes having only one clause, as exact computation is the best choice for them. Afterwards, the tree is revisited bottom-up to assign the best algorithm for a node, given information about the formula and approximation parameters.

In case an exact algorithm is chosen, we allocate the unused ε's and δ's to the remaining nodes of the tree. This is done by re-propagating error and reliability top-down from the root, and updating values of different nodes. The re-propagation will then concern nodes already assigned with approximation algorithms, or remaining nodes of Step 1 having more than two clauses. Note that after this reallocation, we do not update computation methods for assigned nodes.

Though this heuristic optimizes the time of Step 1, as it avoids redeciding assignments of computation algorithms (for already assigned nodes), and consequently avoids bigger number of reallocations over the tree, one may argue that a fairer allocation of values may result on a better choice of computation algorithms, and could produce a less expensive evaluation plan. We adopt the explained approach in our implementation, yet leave this issue open for future investigations on optimizing Step 1.

Step 2: Exploring the tree At the end of this step, we obtain a tree where every node is assigned with an algorithm, and thus weighted by an estimated computation cost. The best evaluation plan is determined by a bottom-up search over the tree: starting from leaves, we test whether the sum of child costs exceeds the parent-node cost, in which case we set the computation at the parent level, and continue the operation for higher nodes. Figure 4.5 illustrates a mock-up example of an evaluation plan produced by the algorithm of Step 2, where computation nodes are marked with flags. Note that the algorithm is correct, since a child node is never more expensive than its parent. However, the sum of children costs can exceed a parent-node cost.

Algorithm The following algorithm implements Step 1 and Step 2 and summarizes our approach for finding the optimal evaluation plan in the deterministic case:

We start with the compilation tree, and a 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 direct parent node, and propagation rules from Section 4.2;
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) We re-propagate 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 bottom-up whether we want to chose to apply the algorithm at a given node (and forget about the algorithms assigned to its descendant) or at one of its ancestors.

4.3.3 Deterministic vs. Randomized Exploration

Both approaches have advantages and drawbacks in terms of the costs involved and quality of results, as each method explores a different space of evaluation plans. The randomized approach is applied to a much bigger space of possible evaluation trees, since different propagations of approximation values can be performed (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). Yet, it is still difficult to set any guarantee on finding a *good-enough* execution plan, especially when the exploration time is bounded and the estimated space is infinitely big. On the other hand, setting a static choice on propagation conditions will produce a much more concise space of plans. However, we still face significant challenges in this second approach, such as problems of reallocations pointed in Step 1 for the deterministic case implemented in ProApproX.

Chapter 5

ProApproX: A Query Engine for Probabilistic XML

5.1 System Overview

ProApproX was designed to be the query engine in a full-fledged probabilistic XML database management system. A first version was released in 2010 (see further in Section 5.3) and was demonstrated at ACM SIGMOD 2011 under the title "ProApproX: a lightweight approximation query processor over probabilistic trees". ProApproX 2.0 integrates a fully redesigned processor with the introduction of most of the features presented in the previous chapters (decomposition of the DNF, cost model, and exploration of possible execution plans). This version is available for download at

`http://www.infres.enst.fr/~souihli/ProApproX2.0.html`.

A demonstration of this latest version is published at CIKM 2012 under the name "Demonstrating ProApproX 2.0: A Predictive Query Engine for Probabilistic XML", and the related scientific contribution is submitted to ICDE 2013, under the name: Optimizing Approximations of Query Lineage in Probabilistic XML.

5.1.1 Implementation

Data representation ProApproX stores a probabilistic document of the PrXML^{cie} model, where probabilities of nodes, expressed with a conjunction of independent events, are recorded in an attribute of these nodes, typically named *"event"* or *"prob"*. To run experiments over document of the local dependency model, we implemented an extended module to translate the document in an equivalent PrXML^{cie} database, according to the following specifications:

- Mutual exclusiveness is described by a parent node, recognized by default with the tag *"mux"* (must be specified otherwise);
- Independency is described by a parent node, recognized by default with the tag *"ind"* (must be specified otherwise);
- Child nodes of *"mux"* and *"ind"* hold an attribute named *"prob"* that records the probability distribution of the corresponding node (the tag of this attribute must be specified otherwise);
- Performing the translation will drop *"mux"* and *"ind"* nodes, keep the *"prob"* attribute, but assign a propositional formula instead of the distribution value. This formula is built to reproduce the dependency relationship that exists between the child nodes (Figure B.3 in Section1.1.1).
- Distributions of Boolean literals used in the new prob attributes, are exported in an external text file during the translation. These values are loaded in a hash table when an associated database is loaded in ProApproX.

XQuery Processor The lineage queries translated in the XQuery language were processed in the first version of ProApproX by Saxon[1], an XQuery 1.0 processor invoked via a supplied Java API.

While running experiments, we noticed that the largest portion of the total run time was recorded by the lineage extraction performed by Saxon. We then replaced the Saxon API by that of eXist-db[2], an open Source Native XML Database management system. We wanted to feature index-based XQuery processing in order to improve the lineage query processing time. Configuring indexes in eXist was not a pleasant experience. Actually there was no straightforward way to set an index for all text nodes or attributes for example. A range index is configured by adding a `<create>` element directly below the root `<index>` node, in the *.xconf* file of a given collection. As shown in the example below, the node to be indexed is either specified through a path or through a *qname* attribute:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
<index>
<create qname="year" type="xs:integer"/>
<create qname="name" type="xs:string"/>
<create qname="title" type="xs:string"/>
<create path="//name" type="xs:string"/>
</index>
</collection>
```

[1] `http://saxon.sourceforge.net/`
[2] `http://exist-db.org/exist/index.xml`

Note however that the obvious solution of specifying `//*` for the path did not work for some reason, when we tried to create path indexes. We did not fully integrate eXist, but had time to run some experiments and record the timing performances. Finally, in ProApproX 2.0, we build the system on top of BaseX[3], where index features were easily turned on, using a simple command:

```
Syntax:    CREATE INDEX [TEXT|ATTRIBUTE|FULLTEXT]
```

Though the lineage extraction time often remains the most important among the other processing times, we noticed that performances of the native XML DBMS BaseX were far better than eXist.

Algorithm Constants he cost model for each algorithm, previously presented in Chapter 3, expresses the complexity of the algorithm via a function $\mathrm{f}_{\mathrm{alg}}$ (for example, $\mathrm{f}_{\mathrm{MulMC}} = \ln \frac{2}{\delta} \times \frac{L}{\ell^2 \varepsilon^2}$). As explained in Chapter 3, we determine $\mathrm{f}_{\mathrm{alg}}$ through a complexity analysis of a given algorithm, taking into account the different inputs (number of literals, number of clauses, approximation parameters etc.). The cost also adds an implementation constant to $\mathrm{f}_{\mathrm{alg}}$. This value is determined in two steps:

1. we run a given algorithm over different sets of input data with increasing sizes (practically, this was done via increasing only the total size of the DNF formula), then record the related run times $\mathrm{time}_{\mathrm{alg}}$;

2. giving that $\mathrm{time}_{\mathrm{alg}} = \mathrm{f}_{\mathrm{alg}} \times c_{\mathrm{alg}}$, we determine the value of the related constant as $c_{\mathrm{alg}} = \frac{\mathrm{time}_{\mathrm{alg}}}{\mathrm{f}_{\mathrm{alg}}}$.

When running measures, the value of c_{alg} converges to a constant value. The latter is then attributed to the cost model of the given algorithm.

Remark. Note that, for a given implementation of an algorithm, constant values are directly related to the current hardware environment (the processor speed, the amount of physical memory that is available, etc.). Ideally, these constants should be determined given this information, i.e., this task should be performed during the set up phase of ProApproX on a given machine.

5.1.2 Query Processing

Figure 5.1 illustrates the architecture of the system:

(1) The user (XPath) query is translated into the lineage query expressed in XQuery language: Section B.2.2;

[3] `http://basex.org/`

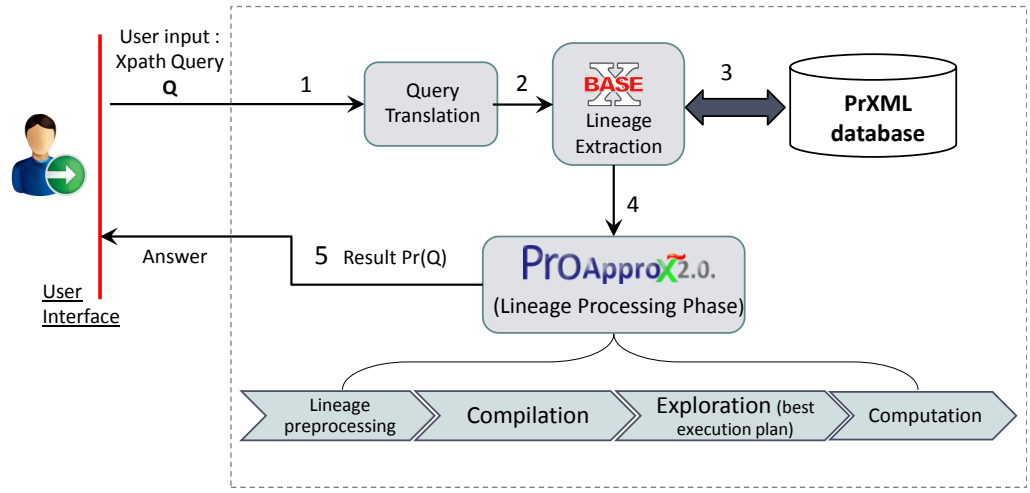


Figure 5.1: The ProApproX 2.0 architecture.

(2, 3) The lineage query is processed by the native DBMS BaseX over the database (PrXML^{cie} p-documents stored as ordinary XML documents with *prob* attributes);

(4) The gathered probabilistic lineage is compiled by ProApproX:

Lineage preprocessing: optimizations over the DNF (removing subsumed or invalid clauses, etc.)
Compilation: Sections 4.1 and 4.2
Exploration: Section 4.3
Computation: following the algorithms set at the computation nodes of the best evaluation tree

(5) The probability of the user query is then displayed.

5.2 ProApproX 2.0

ProApproX 2.0 allows users to query uncertain tree-structured data in the form of probabilistic XML documents. The demonstrated version integrates a fully redesigned query engine that, first, produces a propositional formula that represents the probabilistic lineage of a given answer over the probabilistic XML document, and, second, searches for an optimal strategy to approximate the probability of the lineage. This latter part relies on a query-optimizer–like approach: exploring different evaluation plans for different parts of the formula and predicting the

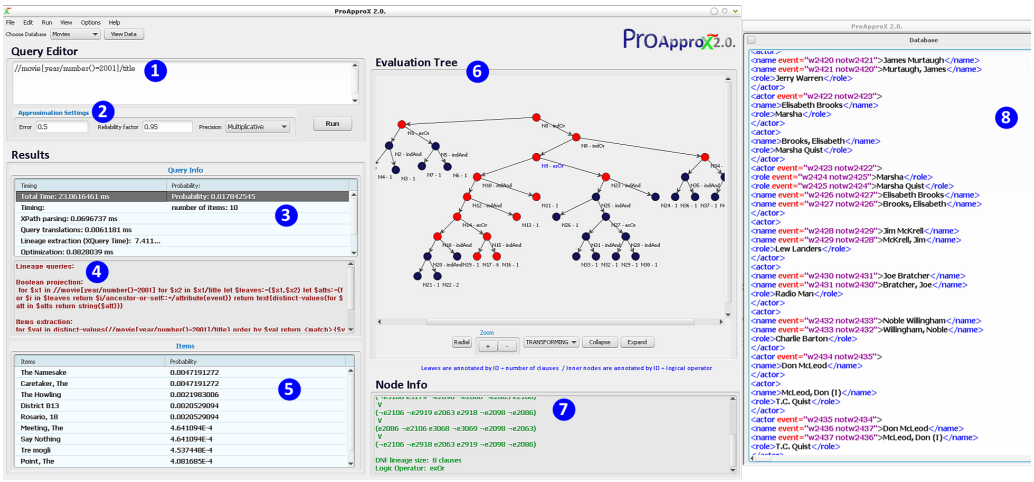


Figure 5.2: The ProApproX 2.0 user interface.

cost of each plan, using a cost model for the various evaluation algorithms. The demonstration presents the graphical user interface of ProApproX 2.0, that allows a user to input an XPath query and approximation parameters, and lists query results with their probabilities; the interface also gives insight into the way the computation is performed, by displaying the compilation of the query lineage as a tree annotated with evaluation operators.

ProApproX 2.0 is implemented in Java as a NetBeans application accompanied by a visual interface that graphically displays information about results as well as processing and computation details. The user can load a $\mathsf{PrXML}^{mux,ind}$ or a PrXML^{cie} database; the latter case requires to give the probability attribute tag to the system and load the events distribution file. Figure 5.2 illustrates the system interface that consists of seven main panels:

(1) The *query editor* is used to input XPath queries on a loaded database, or on the preloaded *movies* [27] and *mondial* [35] probabilistic databases, both kindly provided by the respective authors. For example, the *movies* database features an integration of different instances of a same movie found in more than one source, stored with their respective confidences. One example query is to ask for a movie that features a given actor.

(2) The *approximation settings* section allows to customize parameters for a possible approximation on the result. A combo box offers to choose between an additive or multiplicative approximation, for which an error precision ε and a confidence on the approximated result $1-\delta$ can be customized. These parameters will only be used when the system resorts to a more efficient plan that uses approximations.

(3) In the result panel, the *query information* table displays details about the total time for executing the Boolean projection of the query, and for listing different answers with their computed probabilities. The timing details report slices of time spent at different steps of the evaluation process, namely: the query translation, the BaseX lineage extraction time, performance of optimizations (pre-computation phase), the compilation time, and what was spend on finding and executing the best evaluation plan.

(4) The *lineage query* panel in the result section, displays the translations of the input XPath query into lineage queries. The first one is related to the Boolean projection of the query, and returns a DNF composed of a disjunction between lineages of every matching path in the tree. The second translation produces the list of result items, grouped by their values, each with its DNF lineage.

(5) The *item list* table is then displayed with computed probability values, most probable results appearing first.

(6) The *evaluation tree* panel plots a graphical representation of the evaluation tree where the root node holds the DNF lineage of the query. Inner nodes store independent parts of the initial formula at that given compilation stage, with a logic operator that links the further decompositions at the children level. Ultimately, the leaves will store the last possible decomposition of a previous-level formula. The subtree shown in red is the most efficient execution plan for this query, where red nodes are assigned with computation algorithms. The evaluation of this plan gives the desired probability result.

(7) By picking a node from the graphical representation, its related information will be displayed in the *node info* panel: the propositional formula and logical operator, for internal nodes, or assigned algorithm name, for leaves of the best execution plan.

(8) The *database view* simply displays the currently selected XML database.

A companion video for this demonstration, as well as the full code of ProApproX 2.0, is freely available at `http://www.infres.enst.fr/~souihli/ProApproX2.0.html`.

5.3 Earlier Version

An early version of ProApproX,without any form of cost estimation, was demonstrated in [52]. A user chooses between a variety of query answering techniques, both exact and approximate, and observes the running behavior, pros, and cons,

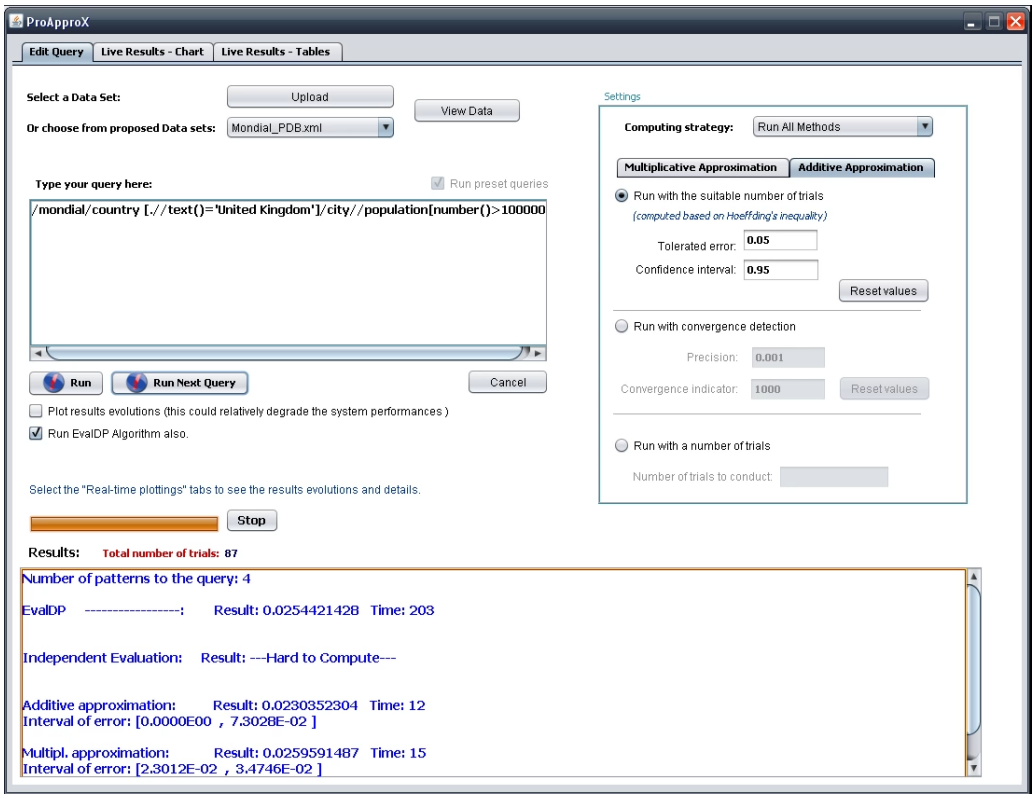


Figure 5.3: Screenshot of the ProApproX main interface

of each method, in terms of efficiency, precision of the result, and data model and query language supported.

We describe here the structure of the user interface for interacting with the system (Figures 5.3 and 5.4), as was demonstrated in [52].

Input The user can select a p-document from a default collection (including the dataset from [35]), or upload an external one. She can specify an XPath query or choose to run a preset query from a proposed list related to the current internal p-document.
Regarding the evaluation method, settings can be custom-made: the user can choose to run one or more algorithms (naïve evaluation, EvalDP, independent exact computation, approximations), or run the default configuration that picks an adequate method for the given query (following a number of simple heuristics). We also give the possibility to personalize the number of samples needed for additive and multiplicative approximations, in one of the following three ways:

Figure 5.4: Plotting the approximation evolution in ProApproX over a given DNF

- by calculating, using Hoeffding's inequality, a suitable number of trials given a tolerated error under a probabilistic guarantee;
- by empirically stopping the sampling once the estimated probability has not deviated more than a given value over a given number of consecutive trials;
- by giving a fixed number of trials.

Output Different results are displayed, namely the probability of the query for each method run, running times, and probabilistic guarantees given by Hoeffding's inequality for approximation techniques.

Additionally, the user can choose to plot the evolution of the estimated probability of approximation techniques and corresponding error intervals, to get a better grasp on the precision obtained by these techniques. When a technique is not applicable to a given p-document or query, the user is also informed of this fact.

Chapter 6

Experiments

In this chapter we evaluate our approach experimentally. We address the following questions:
(1) How does our new query evaluation method compare to the current state-of-the-art probabilistic XML query processing?
(2) How accurate are the results?
(3) How effective is the tree compilation of the DNF formula over simply performing an exact or an approximate computation of the whole formula?

6.1 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. All indexing features of BaseX (in particular, path summaries and text 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 hard drive.

6.1.1 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 [34, 35]. EvalDP is a linear-time bottom-up algorithm for evaluating tree-pattern queries without joins on local dependency p-documents. The queries have

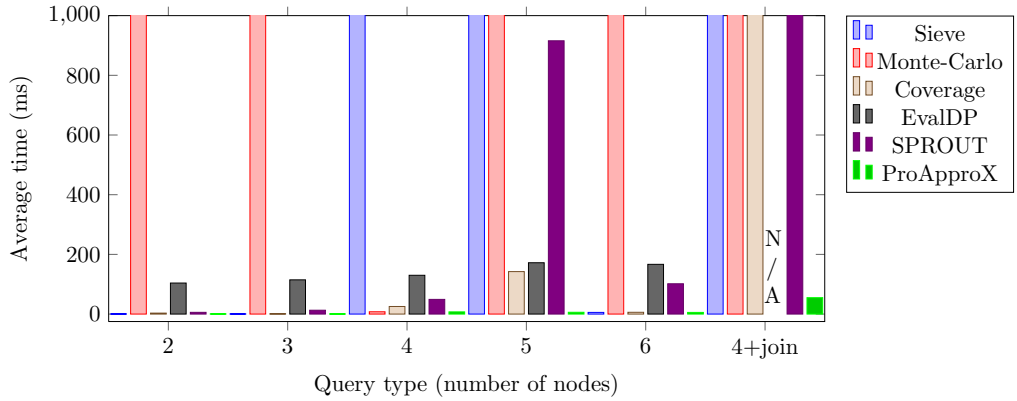


Figure 6.1: Running time of the different algorithms on the MondialDB dataset [34, 35]

between two to six nodes depending on their complexity and selectivity. The p-document, also provided by the authors of [35], 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 [27], from the authors of this work. [27] 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 [57], that produces a local-dependency p-document. The application integrates movie data from a TV guide[1] with that of the Internet Movie Database[2]. As in [27], the query was run on 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 70,000. Specifically, for a given target size number of literals, we iteratively randomly drew clauses in the following manner, until the target size was reached:

   (a) first, draw a random clause size uniformly at random. For four ranges of data size (L_1, L_2, L_3, and L_4) the clause size will be drawn from

[1] `http://www.tvguide.com/`
[2] `http://www.imdb.com/`

different spaces, respectively: $\{1 \,..\, [L_1/(3 \times 3)\}$, $\{1..L_2/(3 \times 10)\}$, $\{1..L_3/(3 \times 100)\}$, and $\{1..L_4/(3 \times 1{,}000)\}$.

(b) Second, uniformly draw positive or negative literals from a fixed set of variables, avoiding contradicting literals and with different random precisions for different sets:

- For DNFs with size $L_1 \leqslant 600$, indices of variables are drawn from $\mathcal{S}_1 = \{1 \dots 600\}$, with a precision of 2 decimals for the random function.
- For DNFs with size $L_2 \leqslant 6{,}000$, indices of variables are drawn from $\mathcal{S}_2 = \{601 \dots 6{,}000\}$, with a precision of 3 decimals for the random function.
- For DNFs with size $L_3 \leqslant 12{,}000$, indices of variables are drawn from $\mathcal{S}_3 = \{6{,}001 \dots 12{,}000\}$, with a precision of 4 decimals for the random function.
- For DNFs with size $L_4 \leqslant 700{,}000$, indices of variables are drawn from $\mathcal{S}_3 = \{12{,}001 \dots 700{,}000\}$, with a precision of 5 decimals for the random function.

The process generates a negated literal with probability 40%.

6.1.2 Methodology

Each query is translated to 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 4.3 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 4.1, 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 $\delta = 95\%$.

Query type	Average DNF size	XQuery	Compilation	Exploration	Evaluation
2	6	96.82%	2.47%	0.19%	0.52%
3	5	98.19%	1.38%	0.08%	0.34%
4	43	97.41%	2.01%	0.34%	0.24%
5	252	64.47%	33.76%	1.43%	0.35%
6	6	99.69%	0.29%	0.02%	0.01%
4+join	3656	61.49%	36.12%	1.95%	0.44%

Table 6.1: Proportion of the time spent on each part of the probabilistic XML query evaluation on the MondialDB dataset for the best tree strategy

6.2 Comparison with EvalDP

We compare with EvalDP, using the implementation kindly provided by the authors of [35]. We show the results of our system, baselines and EvalDP on the 16 queries, averaged by type, in Figure 6.1. The *best tree* strategy of ProApproX has highly competitive execution time, being in all cases under 1s 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 (this is usually the sign that we have 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 the join query, interestingly enough, the best-tree strategy is significantly faster than all 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 data, 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 6.1, 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 being negligible in all cases. Finally recall that EvalDP is neither able to deal with join queries, nor with long-distance dependencies, contrarily to ProApproX.

Accuracy of Results Obviously, a fast algorithm is of no use if the results obtained are not accurate, something to be wary of in the case of ProApproX since it relies on approximation algorithms. To evaluate the quality of results generated

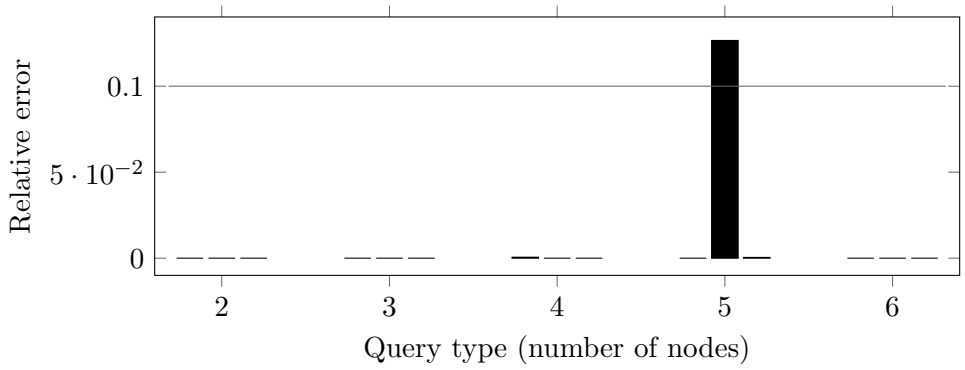


Figure 6.2: Relative error on the probabilities computed by the algorithm on the MondialDB over each non-join query with respect to the exact probability values ($\varepsilon = 0.1$, $\delta = 95\%$)

by ProApproX, we based the comparison on the relative error distance between our results and the exact probability computed by the naive algorithm (except in one case where EvalDP returns a wrong answer, probably due to a bug in the code, where we had to compute the exact answer using the naïve algorithm). We left the join query out of this experiment, since it was too costly to compute its probability in an exact manner. Figure 6.2 is the result on one particular run of the algorithm (since we use sampling, errors tend to be averaged out if we consider multiple runs). The figure 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, but this is in line with what is expected: with a probabilistic guarantee 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.

6.3 Performance over the Movies Dataset

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 than the performances presented by the authors in [27] using Trio, a relational probabilistic DBMS [42]. In Figure 6.3, we plot the performance over the join query Q_3; points missing in the figure mean runtimes greater than 6 seconds. The score for Trio is plotted on the figure, though it is not directly comparable, as we did not reimplement the translation to Trio but took the numbers directly from [27]. Still, it is interesting here to note that Trio has a performance of about 40 seconds on the biggest p-document (19,475 nodes), while ProApproX took around 200 milliseconds.

Figure 6.3 also compares the best-tree strategy to the full-tree strategy (always

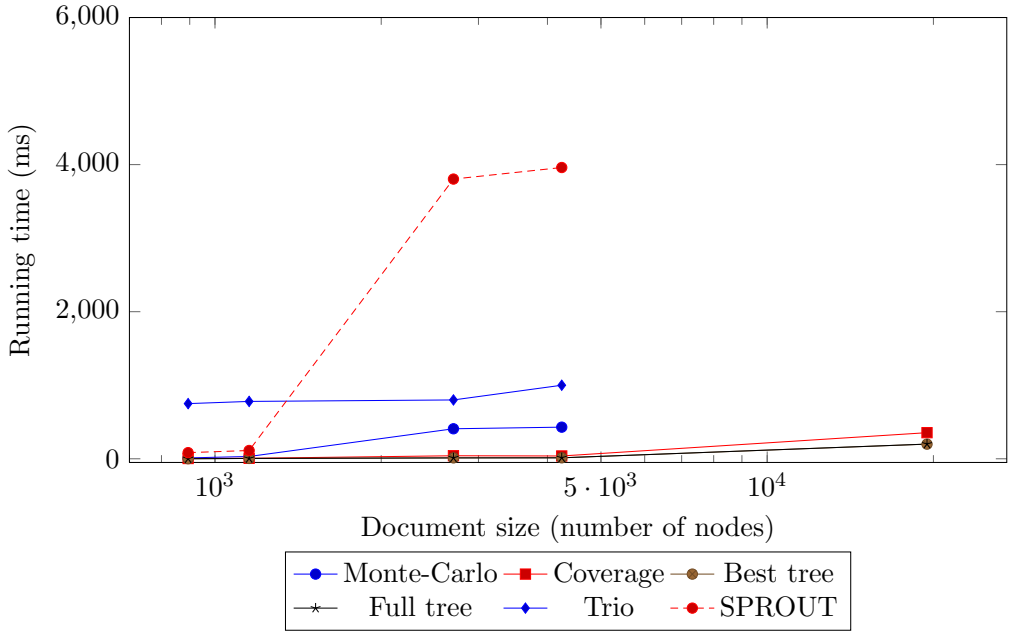


Figure 6.3: Running time of the different algorithms on query Q_3 of the movie dataset [27]; times greater than 6s are not shown and times for Trio are estimated from [27]

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 can hint that the full tree might not have many levels (the clauses might be highly correlated in this lineages). We have seen in Figure 6.1 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).

6.4 Comparison with SPROUT

We also compared with the performances of SPROUT, the query engine in MayBMS [38], over the same DNFs of MondialDB (Figure 6.1) and Movies (Figure 6.3). These DNF formulas are the lineage of the queries over the tree, and they have been encoded as a MayBMS table, with a query returning the lineage probability. We needed to bring some changes to the ICDE'10 version of SPROUT's code[3] [47] to run the *aconf()* function that approximates the probability of a DNF. For example, we had to extend the arity of a number of functions manually on the PostgreSQL

[3] `http://maybms.sourceforge.net/`

catalog to make them accept more variables. We also inserted snippets of code to record the time needed to process a DNF lineage in MayBMS, which should include the compilation and evaluation of the propositional formula excluding all PostgreSQL specific timings, that are of no interest to us here, and that may be high only because of disk storage costs.

Note that if we were able to run the query over an encoding of the probabilistic Mondial database in relational tables, we would report the time spent by MayBMS to extract the DNF lineage (or rather, the time spent by PostgreSQL in extracting the DNF, as MayBMS is built on top of the relational DBMS). With the aim to make the comparison more demonstrative, and report times as they would be experienced by the user, we added the XQuery time to the evaluation of DNFs for SPROUT. We run the experiments on MayBMS using the same approximation parameters in ProApproX (ε=0.1 and 95% for δ). As shown in figures, ProApproX achieves better performances compared to SPROUT. For MondialDB queries, we recorded one order of magnitude improvements on the largest DNFs (queries with 5 nodes, where DNFs have average size of 252). For the last categories of queries (average DNF size equal to 3,656 variables), run times for SPROUT were off scale (1,3,9 and 4 seconds respectively for the four tested queries), while ProApproX never exceeded 100 milliseconds. Also for queries on Movies (Figure 6.3), recorded run times for MayBMS were much longer 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 biggest document (19,475 nodes), SPROUT was running indefinitely, failing to return an estimation of the DNF probability. We do not know whether it is due to a bug in the code, so we stopped the process after almost two hours, without any information on the computed value at that stage.

Actually, SPROUT may perform comparatively badly on these DNFs as it was not really designed for lineages with large clauses as the one we get from PrXML querying, and for queries with many self-joins, as are those we obtain when we encode the lineage probability evaluation into MayBMS.

Finally, note that to estimate the probability of a query, a user calls the function *aconf()* in MayBMS. As this function exclusively uses approximations, this will surely have bad consequences on performance when stricter precisions are required, as the number of trials is directly related to the size of the formula and also to ε and δ, and it is not possible to switch to an exact computation algorithm.

6.5 Performance over Synthetic Data

Figures 6.4 and 6.5 present performances of different evaluation methods: the additive Monte-Carlo, the Self-Adjusting coverage algorithm, evaluation over a fully compiled tree (Full tree), and evaluation over the best execution plan (Best

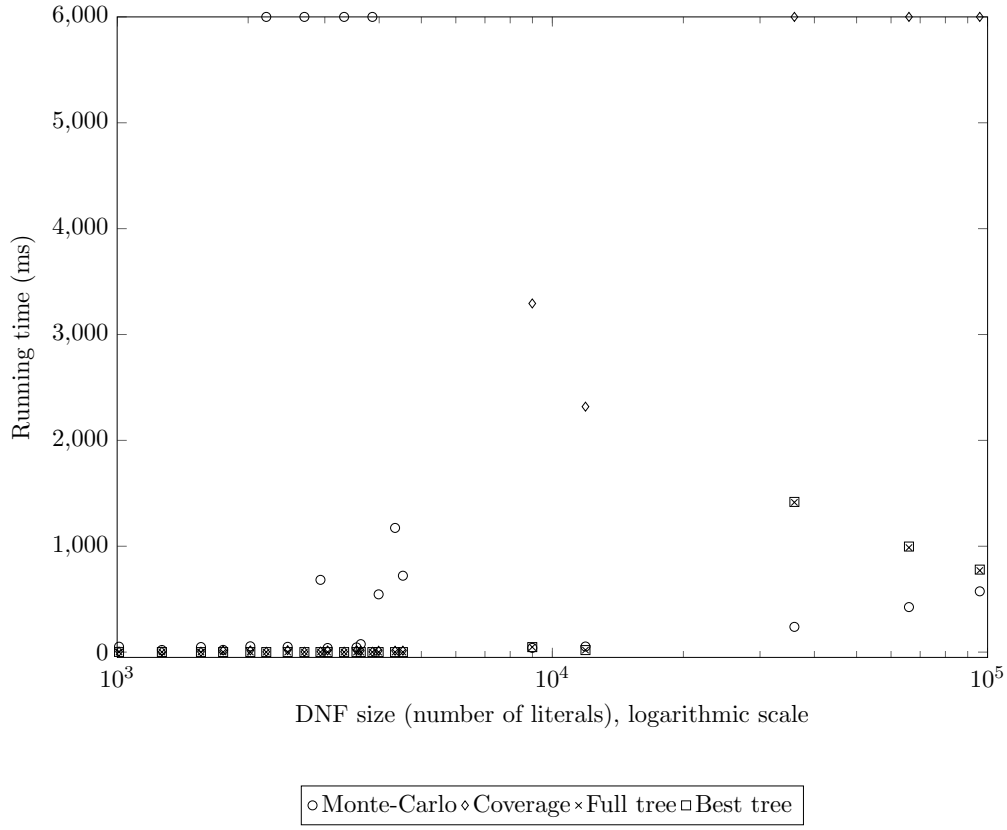


Figure 6.4: Running time of the different algorithms on the synthetic dataset

tree) that ProApproX chooses for a DNF probability computation. Note that times shown on the top line are actually anywhere beyond. As shown through experiments in both figures, 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 random DNFs of Figure 6.5), 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 $\sim 10^{5.6}$ in Figure 6.5), Monte Carlo may record better time,

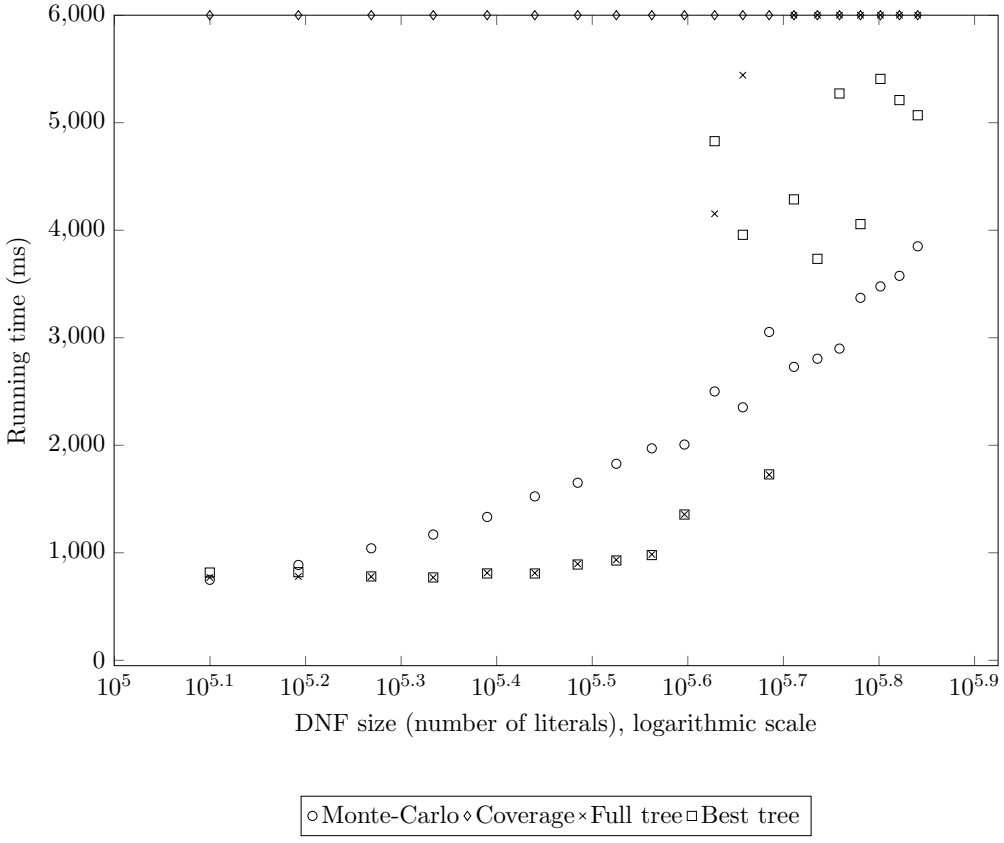


Figure 6.5: Running time of the different algorithms on the synthetic dataset

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 (Figure 6.5), the best-tree strategy does a better job than the full-tree strategy, as was already observed in the previous dataset. 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.

Conclusions

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, and some of the algorithms used, assume 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.

We conclude this thesis by highlighting other potential improvements of interest on the discussed solution, and a number of perspectives in querying probabilistic XML data. Let us start with the improvements:

- It might be valuable to refine our cost model with non-linear functions of

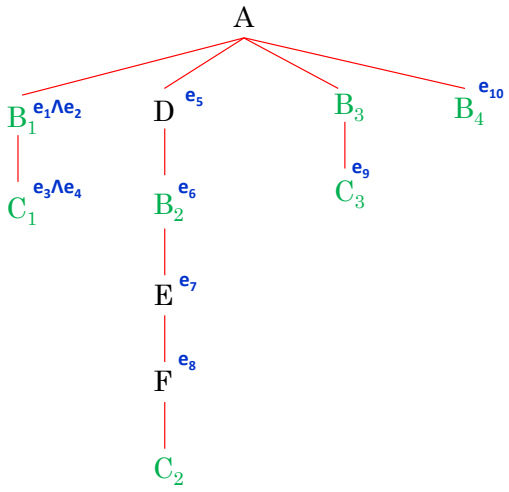
the size of the formulas, to take into account second-order terms in the complexity of the algorithms;

- There is also the issue of exploring the space of evaluation trees in a more intelligent manner, taking into account different possibilities for assigning values of ε and δ to approximated nodes. Though it is always better to apply the naïve algorithm on two children of a node than on the node itself, this observation is not true in general for approximation algorithms, so this requires non-trivial analysis of the cases;

- We might also experiment keeping 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.

Open Issues

Negated Queries A subsequent direction to follow is the support of negated queries. This new class of queries might require effort, yet covers many tractable cases, with possibly a few inherent difficulties in others. Actually, extending our query solution to deal with this class of queries is possible for queries whose probabilistic lineages are in disjunctive normal form. Note however that, generally, negated queries are susceptible to return an exponentially big DNF in the size of the probabilistic data. The technical challenge for negated queries is the lineage extraction, which should follow a special interpretation or semantics over the tree. Consider the following PrXML tree:



When looking for a node B that does not have a descendant node C, the system has also to include a given path from B to a probabilistic node C, as this latter has

a probability of not appearing. The simple pattern query Q:`//B[not(.//C)]` over the preceding tree will then have five possible matches, each with its respective probabilistic lineage that fulfills the negation:
$c_1 = e_1 \wedge e_2 \wedge \neg(e_3 \wedge e_4) = (e_1 \wedge e_2 \wedge \neg e_3) \vee (e_1 \wedge e_2 \wedge \neg e_4)$: this probabilistic path expresses the condition for the occurrence of B_1 and non-occurrence of C_1;
$c_2 = (e_5 \wedge e_6 \wedge \neg e_7)$: occurrence of D and B_2, and non-occurrence of E;
$c_3 = (e_5 \wedge e_6 \wedge e_7 \wedge \neg e_8)$: occurrence of D and B_2, occurrence of E, and non-occurrence of F;
$c_4 = (\neg e_9)$: non-occurrence of C_3;
$c_5 = (e_{10})$: occurrence of B_4.

Sometimes, unfolding the clauses would turn the lineage into a DNF, like the case of the example lineage of Q. In any case, we will need to design a new mechanism for extracting the query lineage. Note also that for cases when a query have more than one negated predicate, new aspects must be taken into consideration when producing the lineage query.

Aggregate Queries With the emergence of probabilistic data in many important application domains, the demand for understanding and processing the ranking and aggregate queries efficiently from the scientific community and beyond is expected to intensify in the near future. The result of a query that makes use of aggregate functions is a set of possible values (for each possible document), each with its probability. A theoretical study that examines the underlying properties associated with the rich semantics of ranking and aggregate queries for probabilistic data, is given in [3], which also introduces continuous distributions (e.g., data provided from sensors [14, 20]). A semantics of lineage for aggregate queries in the relational setting is given in [8], while the problem of compiling this lineage from the query is investigated by R. Fink et al. in [21].

Distributed Computation ProApproX already shows good performance on scalable inputs, as demonstrated empirically through the synthetic experiments. But it is possible to resort to a distributed computation, in case, say, a threshold about the size of an input DNF is reached. In front of a very big number of clauses, it is possible to apply the *independent-or* operator to try and split the DNF over multi-core or distributed architectures. Different values will then be aggregated by an independent disjunction to form the query result. It is also possible to exploit the fact that most evaluation algorithms scale effortlessly to also distribute the probability computation, in particular for sampling.

Distributed PrXML In the context of an autonomous, heterogeneous, decentralized system, uncertainty may originates from imperfect schema matchings, redundancy and contradiction in the information stored by a peers. Because of

the heterogeneous nature of the information shared in such a distributed system, semistructured (i.e., XML) models are favored.

Actually, one of the initial motivation about this work was the *dataring* project launched by Abiteboul and Polyzotis [5], where the authors propose a vision towards building a P2P middleware system that can be used by a community of non-experts, such as scientists, to establish content sharing communities in a declarative fashion upon an infrastructure, called the *dataring*.

Investing research on querying (managing, in general) probabilistic data in a distributed framework, is also a very challenging topic. First, we consider the case in which the peers share processing capability so that a query over the network will gather answers over databases stored by connected peers. One of the problems that arises, even in the case of a deterministic database, is the uncertainty that could result of the fusion of the results. The answer set can then be probabilistically integrated to form a PrXML result. The query can then be rerun over the probabilistic document, to list the different items and their probabilities. In case a peer already hold probabilistic databases, we need to manage an additional degree of uncertainty. Beside the task of answers-sets integration, a more sophisticated procedure must be carefully studied to consider unifying probabilistic instances.

Bibliography

[1] T. Abdessalem, M. L. Ba, and P. Senellart. A probabilistic XML merging tool. In *Proc. EDBT*, 2011.

[2] T. Abdessalem, B. Cautis, and N. Derouiche. Objectrunner: Lightweight, targeted extraction and querying of structured web data. *Proc. VLDB Endowment*, 2010.

[3] S. Abiteboul, T.-H. H. Chan, E. Kharlamov, W. Nutt, and P. Senellart. Aggregate queries for discrete and continuous probabilistic XML. In *Proc. ICDT*, 2010.

[4] S. Abiteboul, B. Kimelfeld, Y. Sagiv, and P. Senellart. On the expressiveness of probabilistic XML models. *VLDB J.*, 2009.

[5] S. Abiteboul and N. Polyzotis. The Data Ring: Community content sharing. In *Proc. CIDR*, 2007.

[6] S. Abiteboul and P. Senellart. Querying and updating probabilistic information in XML. In *Proc. EDBT*, 2006.

[7] M. Ajtai and A. Wigderson. Deterministic simulation of probabilistic constant depth circuits (preliminary version). In *FOCS*, 1985.

[8] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *Proc. PODS*, 2011.

[9] L. Antova, C. Koch, and D. Olteanu. Query language support for incomplete information in the MayBMS system. In *Proc. VLDB*, 2007.

[10] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *Proc. IEEE*, 1992.

[11] M. Benedikt, E. Kharlamov, D. Olteanu, and P. Senellart. Probabilistic XML via Markov chains. *Proc. VLDB Endowment*, 2010.

[12] M. Benedikt and C. Koch. XPath leashed. *Proc. ACM*, 2008.

[13] J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proc. SIGMOD*, 2005.

[14] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proc. SIGMOD*, 2003.

[15] S. Cohen, B. Kimelfeld, and Y. Sagiv. Incorporating constraints in probabilistic XML. *Proc. ACM*, 2009.

[16] S. Cohen, B. Kimelfeld, and Y. Sagiv. Running tree automata on probabilistic XML. In *Proc. PODS*, 2009.

[17] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross. An optimal algorithm for monte carlo estimation. *SIAM J.*, 2000.

[18] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 2007.

[19] N. N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *Proc. PODS*, 2007.

[20] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. VLDB*, 2004.

[21] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. *Proc. VLDB Endowment*, 2012.

[22] R. Fink and D. Olteanu. On the optimal approximation of queries using tractable propositional languages. In *Proc. ICDT*, 2011.

[23] R. Fink, D. Olteanu, and S. Rath. Providing support for full relational algebra in probabilistic databases. In *Proc. ICDE*, 2011.

[24] R. Gilleron, F. Jousse, I. Tellier, and M. Tommasi. Xml document transformation with conditional random fields. In *INEX*, 2006.

[25] K. D. Heidtmann. Improved method of inclusion-exclusion applied to k-out-of-n systems. *Proc. IEEE*, 1982.

[26] W. Hoeffding. Probability inequalities for sums of bounded random variables. *J. American Statistical Association*, 1963.

[27] E. Hollander and M. van Keulen. Storing and querying probabilistic XML using a probabilistic relational DBMS. In *Proc. MUD*, 2010.

[28] J. Huang. Design and implementation of the SPROUT query engine for probabilistic databases. Master's thesis, University of Oxford, 2009.

[29] J. Huang, L. Antova, C. Koch, and D. Olteanu. MayBMS: a probabilistic database management system. In *Proc. SIGMOD*, 2009.

[30] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 1984.

[31] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *FOCS*, 1983.

[32] R. M. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Algorithms J.*, 1989.

[33] E. Kharlamov and P. Senellart. Modeling, querying, and mining uncertain XML data. In A. Tagarelli, editor, *XML Data Mining: Models, Methods, and Applications*. IGI Global, 2011.

[34] B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv. Query efficiency in probabilistic XML models. In *Proc. SIGMOD*, 2008.

[35] B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv. Query evaluation over probabilistic XML. *VLDB J.*, 2009.

[36] B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic XML. In *Proc. VLDB*, 2007.

[37] B. Kimelfeld and P. Senellart. Probabilistic XML: Models and complexity. In Z. Ma, editor, *Advances in Probabilistic Databases for Uncertain Information Management*. Springer-Verlag, 2012. To appear.

[38] C. Koch. MayBMS: A system for managing large uncertain and probabilistic databases. In *Managing and Mining Uncertain Data*. Springer-Verlag, 2008/9.

[39] M. Luby and B. Velickovic. On deterministic approximation of DNF. *Algorithmica J.*, 1996.

[40] S. Maniu, B. Cautis, and T. Abdessalem. Building a signed network from interactions in wikipedia. In *DBSocial*, 2011.

[41] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*. Springer, 1998.

[42] 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 *Proc. CIDR*, 2007.

[43] A. Nierman and H. V. Jagadish. ProTDB: Probabilistic data in XML. In *Proc. VLDB*, 2002.

[44] D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *Proc. SUM*, 2008.

[45] D. Olteanu and J. Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *Proc. SIGMOD*, 2009.

[46] D. Olteanu, J. Huang, and C. Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, 2009.

[47] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *Proc. ICDE*, 2010.

[48] C. Re, N. N. Dalvi, and D. Suciu. Query evaluation on probabilistic databases. *Proc. IEEE*, 2006.

[49] C. Re, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. ICDE*, 2007.

[50] P. Senellart and S. Abiteboul. On the complexity of managing probabilistic XML data. In *Proc. PODS*, 2007.

[51] P. Senellart, A. Mittal, D. Muschick, R. Gilleron, and M. Tommasi. Automatic wrapper induction from hidden-web sources with domain knowledge. In *WIDM*, 2008.

[52] P. Senellart and A. Souihli. ProApproX: a lightweight approximation query processor over probabilistic trees. In *Proc. SIGMOD*, 2011.

[53] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *Proc. SIGMOD*, 2008.

[54] A. Souihli. Efficient query evaluation over probabilistic XML with long-distance dependencies. In *Proc. EDBT/ICDT*, 2011.

[55] D. L. Vail, M. M. Veloso, and J. D. Lafferty. Conditional random fields for activity recognition. In *AAMAS*, 2007.

[56] L. G. Valiant. The complexity of enumeration and reliability problems. *j. SIAM*, 1979.

[57] M. van Keulen and A. de Keijzer. Qualitative effects of knowledge rules and user feedback in probabilistic data integration. *VLDB J.*, 2009.

[58] M. van Keulen, A. de Keijzer, and W. Alink. A probabilistic XML approach to data integration. In *Proc. ICDE*, 2005.

Appendix A

This appendix reports the queries tested on ProApproX. First set of queries was kindly provided by Kimelfeld, Kosharovsky, and Sagiv, the authors of EvalDP [35]. The second set reproduces queries used in [27] about querying probabilistic XML using Trio, a probabilistic relational DBMS.

Mondial Database Queries

/* 2 nodes */
Q1:
```
/mondial//text()[.='Barcelona']
```
Lineage Query:
```
for $a in /mondial//text()[.='Barcelona']
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```
Q2:
```
/mondial//text()[.='UN']
```
Lineage Query:
```
for $a in /mondial//text()[.='UN']
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```
Q3:
```
/mondial//text()[.='Kilimanjaro']
```
Lineage Query:
```
for $a in /mondial//text()[.='Kilimanjaro']
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

/* 3 nodes */
Q4:
```
/mondial//name[text()='Barcelona']
```
Lineage Query:

```
for $a in /mondial//name[text()='Barcelona']
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q5:

```
/mondial/organization//text()[.='UN']
```

Lineage Query:

```
for $a in /mondial/organization//text()[.='UN']
let $atts:=(for $i in $a
return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q6:

```
/mondial/mountain//text()[.='Kilimanjaro']
```

Lineage Query:

```
for $a in /mondial/mountain//text()[.='Kilimanjaro']
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

/* 4 nodes */

Q7:

```
/mondial/country/attpopulation[number(text())>50000000]
```

Lineage Query:

```
for $a in /mondial/country/attpopulation [number(text())>50000000]
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q8:

```
/mondial/country[.//text()='Calgary'][.//text()='Alberta']
```

Lineage Query:

```
for $a in /mondial/country
for $b in $a//text()[.='Calgary']
for $c in $a//text()[.='Alberta']
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q9:

```
/mondial/mountain/attheight[number(.//text())>5000]
```

Lineage Query:

```
for $a in /mondial/mountain/attheight [number(.//text())>5000]
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

/* 5 nodes */

Q10:

```
/mondial//province[.//text()='Berkshire']//population[number(text())>100000.0]
```
Lineage Query:
```
for $a in /mondial//province
for $b in $a//text() [.='Berkshire']
for $c in $a//population [number(text())>100000.0]
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```
Q11:
```
/mondial/country[.//text().='United Kingdom']/city//*[number(text())>100000.0]
```
Lineage Query:
```
for $a in /mondial/country
for $b in $a//text() [.='United Kingdom']
for $c in $a/city//*[number(text())>100000.0] let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```
Q12:
```
/*/country//city//@type [text()='sea']
```
Lineage Query:
```
for $a in /*/country//city//atttype [text()='sea']
let $atts:=(for $i in $a return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

/* 6 nodes */
Q13:
```
/mondial/country/*/text()[.='Muslim']/population [number(text())>10000000.0]
```
Lineage Query:
```
for $a in /mondial/country/*
for $b in $a/text()[.='Muslim']
for $c in $a/population [number(text())>10000000.0]
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```
Q14:
```
/mondial/country[.//text()='Indonesia']
[.//text()='Muslim']/@population[number(text())>10000000]
```
Lineage Query:
```
for $a in /mondial/country
for $b in $a//text() [.='Indonesia']
for $c in $a//text() [.='Muslim']
for $d in $a/attpopulation [number(text())>10000000]
let $feuilles:=($b,$c,$d)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q15:
```
/mondial/*/province/city[.//text()='Barcelona']//*[number(text())>1000000.0]
```
Lineage Query:
```
for $a in /mondial/*/province/city
for $b in $a//text()[.='Barcelona']
for $c in $a//* [number(text())>1000000.0]
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

/* Join queries */
Q16:
```
/mondial//country[.//name=.//attname]
```
Lineage Query:
```
for $a in /mondial//country
for $b in $a//name
for $c in $a//attname
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
where $c=$b
return text{distinct-values(for $att in $atts return string($att))}
```

Q17:
```
/mondial//country[.//attcapital=.//attid]
```
Lineage Query:
```
for $a in /mondial//country
for $b in $a//attcapital
for $c in $a//attid
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
where $c=$b
return text{distinct-values(for $att in $atts return string($att))}
```

/* Other Queries */
Q18:
```
mondial//country[.//text()='Calgary'][//*]
```
Lineage Query:
```
for $a in mondial//country
for $b in $a//*[text()='Calgary']
for $c in $a//*
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{ distinct-values(for $att in $atts return string($att))}
```

Q19:

```
mondial//country[.//text()='Calgary'][.//number()>20000000]
```

Lineage Query:

```
for $a in mondial//country
for $b in $a//*[text()='Calgary']
for $c in $a//*[number()>20000000]
let $feuilles:=($b,$c)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{ distinct-values(for $att in $atts return string($att))}
```

Movie Database Queries

Q1:

```
//movie[title='District B13']/year/text()
```

Lineage Query:

```
for $a in //movie[title='District B13']
for $b in $a/year/text()
let $feuilles:=($b)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q2:

```
//movie[actors/actor/name='Brooke Smith']/title/text()
```

Lineage Query:

```
for $a in //movie[actors/actor/name='Brooke Smith']
for $b in $a/title/text()
let $feuilles:=($b)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Q3:

```
//movie[year=//movie[actors/actor/name='David Belle']/year]/title/text()
```

Lineage Query:

```
for $a in //movie[year=//movie[actors/actor/name='David Belle']/year]
for $b in $a/title/text()
let $feuilles:=($b)
let $atts:=(for $i in $feuilles return $i/ancestor-or-self::*/attribute(event))
return text{distinct-values(for $att in $atts return string($att))}
```

Appendix B

Résumé en français

Introduction

De nombreuses applications, telles que l'intégration des données incertaines [57], la gestion des entrepôts de données XML [50], les systèmes probabilistes de contrôle de versions [1], ou l'extraction automatique de données sur le Web [2, 24, 51] nécessitent une gestion de l'incertitude caractérisant ce type d'applications. Dans le dernier exemple cité, le système pourrait avoir une certaine confiance dans l'information extraite, qui peut être exprimée sous forme d'une probabilité indiquant la chance que l'information existe ou est vraie (par exemple, en y attachant une variable aléatoire conditionnelle [55]) ou par l'affectation d'une valeur de confiance arbitraire. Une mesure de cette incertitude peut être induite à partir de la fiabilité des ressources, la qualité des procédures de mapping des données, etc. Souvent, l'information est décrite de manière semi-structurée, car la représentation au moyen d'une hiérarchie de nœuds est adéquate, en particulier lorsque la source est déjà représentée sous cette forme (par exemple XML ou HTML). Une manière possible, parmi les plus naturelles, pour représenter cette incertitude est d'utiliser les bases de données probabilistes, et XML probabiliste [37] en particulier.

Les *documents probabilistes*, ou *p-documents* [4, 34], sont un système de représentation générale des bases de données XML probabilistes, se basant sur une représentation XML, associant des nœuds ordinaires et probabilistes. Ces derniers définissent le processus de génération d'une instance XML aléatoire suivant les distributions spécifiées au niveau de chaque nœud. Le modèle est une représentation compacte et complète d'un espace probabiliste de documents (c.-à-d., un ensemble fini mais exponentiellement grand de *mondes possibles*, chacun avec une probabilité donnée). Étant donné un document probabiliste, nous proposons dans cette thèse d'étudier la manière de l'interroger efficacement.

Nous nous intéressons donc à l'interrogation de documents XML probabilistes,

en utilisant XPath, un langage de requêtes qui permet de naviguer dans l'arborescence XML et de sélectionner des nœuds par une variété de critères et de prédicats. Dans ce qui suit, nous discutons une solution pour traiter efficacement un sous-ensemble de requêtes XPath, défini par les requêtes *tree-pattern*, éventuellement avec *jointure* (Section B.2).

Étant donné un document XML probabiliste, la confiance liée à une réponse y retournée par une requête sur ce document est la probabilité combinée de tous les mondes possibles dans lesquels y se produit. Ce calcul est l'opération clé qui distingue les bases de données probabilistes des bases de données traditionnelles. Cependant, le calcul de la probabilité exacte d'une réponse se produisant dans un résultat de requête sur un p-document est #P-difficile [35]. En terme de complexité dans les données, il existe un schéma d'approximation polynomial randomisé [35] pour estimer la confiance d'un résultat donné d'une requête Q : nous nous proposons donc de nous concentrer sur l'estimation efficace de cette quantité.

En effet, pour de nombreuses applications, une bonne estimation de la valeur de probabilité est suffisante, c.-à-d., une approximation *à un facteur multiplicatif* de la probabilité correcte, *à haute confiance*. Les approximations *à un facteur additif* sont de moindre intérêt, car c'est assez difficile de distinguer par exemple entre les probabilités de 10^{-2} et 10^{-5}.

En suivant les propositions de [35, 50], nous réduisons le problème de l'approximation de la probabilité d'un résultat d'une requête Q sur un p-document $\mathscr{P}$ à l'approximation de la probabilité d'une formule propositionnelle φ sous forme normale disjonctive (FND). Nous passons d'abord par une ré-écriture de la première requête Q (XPath) en une requête XQuery Q' qui retourne, pour chaque match de Q sur le document déterministe $\mathscr{P}$, la conjonction des événements conditionnant ce match (probabilités étiquetées associées aux nœuds probabilistes du p-document). La requête traduite est alors évaluée par un processeur XQuery standard, permettant l'utilisation de toutes les techniques standard d'indexation XML. Cette étape peut être effectuée en temps polynomial en la taille de $\mathscr{P}$. La disjonction des conjonctions de tous les événements associés aux différentes réponses de la requêtes constitue la *provenance* propositionnelle φ de la requête. Notons que le calcul de la probabilité d'une formule propositionnelle revient à la recherche des assignations satisfaisant la formule, un problème connu pour être #P-complet [56].

En contraste avec les travaux existants [34–36] qui proposent des algorithmes pour un nombre de sous-cas traitables, et soulignent la complexité du problème, nous considérons dans ce travail une forme très générale de p-documents (impliquant des corrélations arbitraires entre les nœuds de l'arborescence) et une plus large classe de requêtes (des requêtes tree-pattern avec jointures, avec des résultats

extensibles à la classe encore plus général de requêtes dites *localement monotones* [50]), visant par ailleurs une solution efficace et pratique pour l'interrogation de p-documents. ProApproX est un système d'interrogation de données XML probabilistes qui implémente un certain nombre de solutions discutées dans cette thèse. Le système a pour caractéristique de se baser sur plus d'un algorithme pour évaluer la probabilité de la formule propositionnelle qui représente la provenance probabiliste d'un résultat d'une requête donnée. Le système repose sur un modèle de coût des différents algorithmes d'évaluation possibles afin de choisir le meilleur algorithme d'approximation pour une formule donnée. De plus, la formule est compilée en un plan d'évaluation, différents algorithmes d'évaluation peuvent être utilisés sur des sous-parties différentes de la formule, et le coût global de l'ensemble du plan est estimé. Comme pour le cas des optimiseurs de requêtes régulières, l'espace de plans d'évaluation (pour une garantie d'approximation spécifiée par l'utilisateur) est exploré afin de trouver le plan de coût optimal.

Pour résumer, nous mettons en évidence les principales contributions de notre thèse, à travers lequel ProApproX révèle son originalité :

1. Le soutien d'un large éventail de requêtes XPath sur un modèle de données plus général que ceux considérés par les systèmes actuels de gestion d'XML probabilistes (Chapitre 1) ;

2. Un modèle de coût pour une variété d'algorithmes de calcul ou d'estimation de probabilité d'une formule en FND, suivant lequel la sélection de l'algorithme le plus efficace est effectuée (Chapitre 3) ;

3. La simplification du processus de calcul via une décomposition de la provenance probabilistes en parties indépendantes d'évaluation (Section 4.1)

4. La possibilité de spécifier une erreur ε et une confiance δ arbitraires pour l'approximation désirée, avec une mise en place rigoureuse de mécanismes de propagation d'erreur et de garanties entre les différentes unités de calcul (Section 4.2) ;

5. Une exploration de l'espace des plans d'évaluations possibles suivant le modèle de coût proposé (Section 4.3) ;

6. Un système fonctionnel implémenté, avec une interface de démonstration, représentant une contribution majeure de la présente thèse (Chapitre 5).

Le chapitre 1 introduit les préliminaires en présentant les différents modèles d'XML probabiliste, la syntaxe, la sémantique des classes de requêtes *tree-pattern* adressées dans ProApproX, la provenance probabiliste, ainsi qu'une formulation propre de la problématique derrière l'interrogation des arbres probabilistes. Le chapitre 2 présente l'état de l'art du problème d'interrogation des bases de données

probabilistes, en présentant une collection de travaux et système des plus pertinents tant pour le modèle relationnel que le semi-structuré (nous nous concentrons sur les techniques de calcul de probabilité déployés dans ces systèmes). Avant de conclure, un dernier chapitre présente une évaluation expérimentale approfondie du système proposé, et compare les performances de notre moteur d'interrogation avec l'état de l'art.

B.1 XML probabiliste

Nous rappelons ici quelques notions de base concernant les bases de données XML probabilistes (ou PrXML) comme un modèle de données incertaines. Le modèle présenté ici a été introduit et étudié dans [4, 34, 35] comme une généralisation des modèles déjà existants [6, 43, 58]. Nous modélisons les documents XML comme des arbres étiquetés, non classés et ordonnés. Ne pas prendre en compte l'ordre entre les nœuds frères dans un document XML est une hypothèse courante, mais non essentielle. La même modélisation peut s'appliquer aux arbres ordonnés, sans changement majeurs à la théorie.

B.1.1 Modèles

Un *document XML probabiliste* (ou *p-document*) est similaire à un document XML, à la différence qu'il dispose de deux types de nœuds : ordinaires et de distribution. Ces derniers sont des nœuds fictifs qui spécifient la façon dont leurs enfants peuvent être choisis au hasard. Compte tenu des critères de dépendance entre les éléments, on distingue deux modèles de représentation de données :

- Dans le modèle de *dépendance locale* [43, 58] (nommé $\mathsf{PrXML}^{mux,ind}$ en [4]), les choix effectués pour les nœuds sont indépendants ou dépendants localement, c'est à dire, un nœud de distribution choisi un ou plusieurs enfants indépendamment d'autres choix faits ailleurs dans l'arbre. La Figure B.1 illustre un arbre probabiliste du modèle de dépendance locale. Cet exemple peut être le résultat d'une intégration d'un certain nombre de répertoires contenant des informations sur les instances de *personnes*. Par exemple, les probabilités peuvent représenter les statistiques recueillies lors de la recherche d'une information dans un ensemble de répertoires intégrés. Un premier numéro de téléphone `3333` lié à la personne nommée `Chris` a été trouvé avec une probabilité indépendante de celles d'autres noeuds, ayant une valeur de 0,1 (ou 10%). L'indépendance de cette répartition est indiquée par le nœud fictif *ind* dans l'abstraction de $\mathsf{PrXML}^{mux,ind}$. Le nœud *mux* dans le même exemple indique que ces descendants directs sont mutuellement exclusifs, comme pour les exemples d'adresses possibles trouvées pour `Chris`.

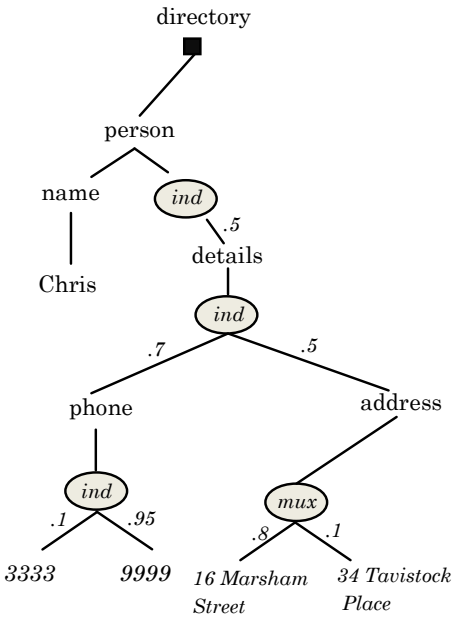


FIGURE B.1 – PrXML^{mux,ind} - Modèle de *dépendance locale*

- Dans le modèle de *dépendance globale* [6] (PrXML^{cie} dans [4]), la génération d'un nœud donné pourrait dépendre des conditions de génération (les choix probabilistes) associées à d'autres parties de l'arbre. Ce modèle, plus général que le modèle de dépendance locale, utilise exclusivement le nœud de distribution *cie* (*Conjonction d'Événements Indépendants*) : les nœuds de ce type sont associés à une conjonction de variables booléennes aléatoires (et possiblement leur négation) et indépendantes $e_1 \dots e_m$ appelées *événements* ; chaque événement a une probabilité globale et indépendante $\Pr(e_i)$ que e_i est vraie. Notons que les différents nœuds *cie* peuvent partager des événements communs, donc les choix peuvent être corrélés.

  L'exemple PrXML^{cie} de la figure B.2 a le pouvoir d'exprimer les mêmes conditions utilisées que dans le PrXML^{mux,ind} de la figure B.1, en utilisant les variables booléennes indépendantes associées avec leurs probabilités respectives. L'inconsistance entre les instances d'adresses est désormais exprimée par des formules propositionnelles impliquant des variables booléennes et leur négation.

  En outre, la dépendance entre différents noeuds `city` éloignés dans cet arbre est possible grâce à l'utilisation d'une même variable. Cela peut être liées à

une sémantique arbitraire généralisée sur l'ensemble de la base de données et qui unifie les choix relatifs à toute instance `city` de la base.

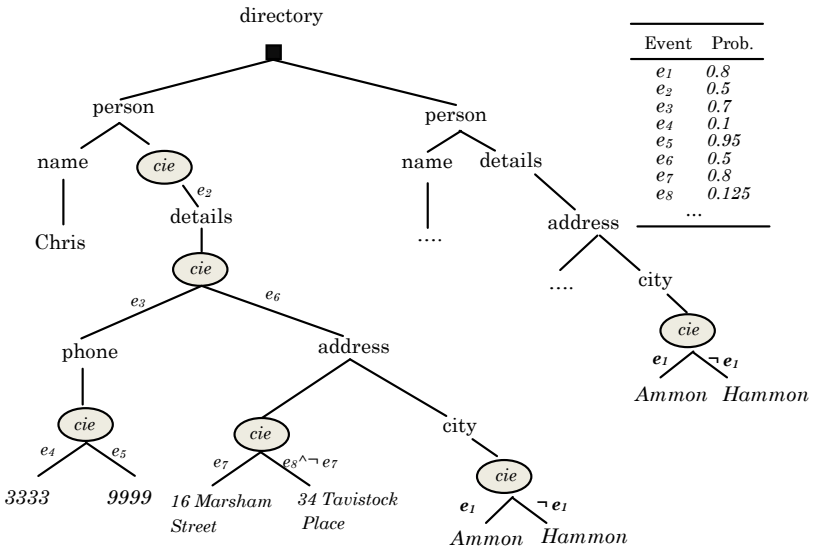


FIGURE B.2 – PrXML^{cie} - Modèle de *dépendance globale*

B.1.2 De PrXML^{mux,ind} à PrXML^{cie}

Comme il est indiqué dans [4], Il existe un algorithme linéaire pour traduire une base de donnée du modèle de dépendance locale en un document PrXML^{cie} (Figure B.3). La probabilité relative au nœud $Child_3$ est associée à la variable booléenne e_2 dans l'arbre PrXML^{cie} équivalent. $Child_1$ et $Child_2$ sont mutuellement exclusifs dans la représentation PrXML^{mux,ind} à travers l'introduction d'un nœud parent *mux*. La traduction est effectuée par la construction de deux formules propositionnelles inconsistantes, ayant respectivement pour probabilités 0,3 et 0,5. Il suffit dans ce cas simple de créer une nouvelle variable booléenne e_1 ayant la probabilité 0,3 et l'associer au nœud $Child_1$, puis trouver une formule pour $Child_2$ ayant la probabilité 0,5 et contenant une négation de e_1.

Nous utilisons ces résultats pour traduire efficacement les bases de données du modèle de dépendance locale en p-documents reposant exclusivement sur des nœuds de distribution *cie*. Nous considérons donc le modèle de dépendance globale dans le reste du travail présenté dans cette thèse.

La *sémantique* d'un p-document est une distribution de probabilité sur un un ensemble de documents possibles, définie par le procédé suivant pour la cas de dépendance globale (Figure B.4). Tout d'abord, tirer au hasard une valeur

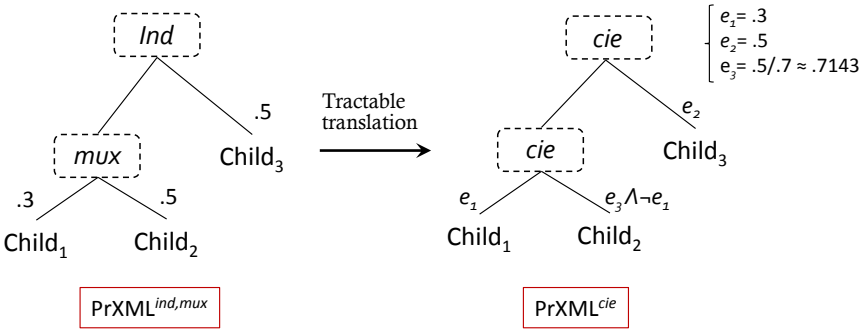


FIGURE B.3 – De PrXML^{mux,ind} à PrXML^{cie}

de vérité pour chacune des varaiables e_i suivant la distribution respective $\Pr(e_i)$. Par la suite, nous retirons tout nœud de distribution évalué à « faux » suivant cette assignation. Les nœuds descendants des nœuds de distribution retirés sont à leurs tours retirés ; et les nœuds restants ordinaires sont connectés à leur plus proche ancêtre ordinaire commun, ce qui résulte en la génération d'une instance aléatoire de document ordinaire. La probabilité de ce document est celle de toutes les assignations de valeurs de vérité qui réalise sa génération.

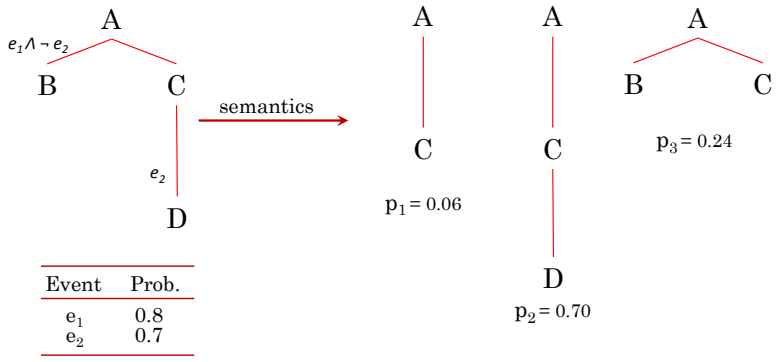
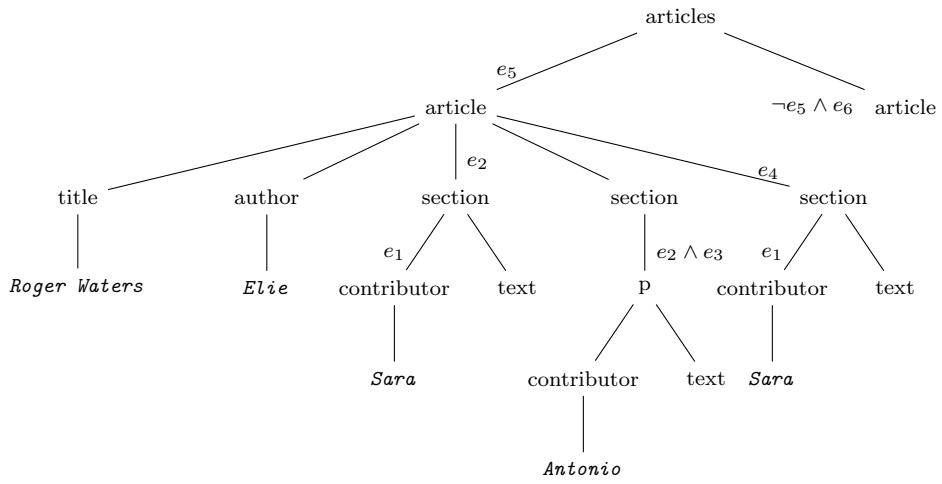


FIGURE B.4 – Développement/déroulement d'un document PrXML^{cie} : mondes possibles

A titre d'exemple, la figure B.5 présente un fragment d'un document XML probabiliste décrivant un article Wikipedia donné comme une fusion (probabiliste) de toutes ses révisions, exemple reproduit (avec élaboration) depuis [1]. Pour faciliter la présentation, les nœuds *cie* sont représentés par une simple annotation des arrêts de l'arbre XML, se composant de conjonctions de littéraux. Par exemple, le premier et le troisième éléments « contributor » de l'arbre apparaissent sous *cie*



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
...	

FIGURE B.5 – Représentation en arbre d'un p-document avec dépendances globales

nœuds qui les retiennent seulement si e_1 est vrai, le tout premier « article » est maintenu si et seulement si e_5 est vrai, etc.

B.2 Interrogation des bases de données XML probabilistes

Le choix entre un modèle PrXML plus concis et plus expressif, et l'efficacité de l'évaluation de requêtes est un compromis clair et naturel. Les auteurs de *EvalDP*, un algorithme linéaire pour l'évaluation de requêtes simples en tree-pattern sur des p-documents à dépendances locale [35], affirment que :

> « Les requêtes ne peuvent être évaluées exactement et efficacement que lorsque les nœuds de distributions sont indépendants en termes probabilistes. »

Le traitement de requêtes tree-pattern sur $\mathsf{PrXML}^{mux,ind}$ est linéaire [35], et (comme on le verra plus tard) exponentiel sur les nœuds cie. Dans cette thèse, nous étudions

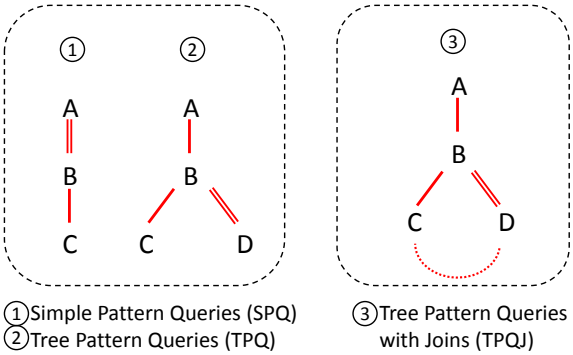


FIGURE B.6 – Types de requêtes supportées (ProApproX)

la complexité d'une classe spécifique de requêtes XPath (plus générale que les requêtes tree-pattern simples) sur les bases de données PrXML^{cie}, sans restriction sur l'expression de dépendance entre n'importe quels ensemble de nœuds dans la base entière, et d'analyser les principaux défis derrière une solution exploitable et pratique, fournissant des résultats avec une bonne précision.

B.2.1 Requêtes étudiées

Le langage de requete que nous considérons dans ce travail concerne les requêtes tree-pattern avec jointure [6]. Une requête tree-pattern avec jointure est définie par un modèle d'arbre (un arbre dont les arcs sont étiquetés avec soit *child* ou *descendant* et dont les nœuds sont étiquetés, éventuellement par un métacaractère) ainsi qu'un ensemble de *valeurs de jointures* imposant que deux nœuds du modèle d'arbre sont appariés à des nœuds d'une même valeur. Une réponse à la requête sur un document déterministe doit avoir une structure en tree-pattern qui respecte les contraintes suivantes :

(i) la racine est mappée à la racine ;

(ii) les arrêtes des nœuds enfants sont mises en correspondance avec les arrêtes de l'arbre ;

(iii) les éléments descendants sont mis en correspondance avec une séquence d'arrêtes des éléments ≪ child ≫ de l'arbre ;

(iv) les étiquettes autres que les métacaractères sont conservées ;

(v) les nœuds d'une condition de jointure doivent avoir la même étiquette.

Comme dans [6], nous considérons chaque réponse comme le sous-arbre minimal contenant tous les nœuds d'un document qui sont associés à un nœud de la requête. Les requêtes sont exprimées en utilisant la syntaxe XPath (notons que certaines requêtes en tree-pattern avec jointure ne peuvent pas être exprimées en XPath 1.0, mais peuvent l'être en XPath 2.0).

Exemple 1. *Étant donné le p-document de la figure B.5, on peut rechercher tous les contributeurs à un article donné en utilisant la requête tree-pattern Q_1 suivante :*

```
//article[title='Roger Waters']//contributor
```

De même, la requête tree-pattern Q_2 utilisant un prédicat de jointure :

```
/articles/article[author=.//contributor]
```

cherche des articles où l'auteur apparaît parmi les contributeurs.

Bien que nous nous concentrons sur les requêtes tree-pattern avec jointures pour la simplicité de présentation, un traitement similaire peut être appliqué à une classe plus large de requêtes, à savoir les requêtes dites *locally monotone* [50], qui comprennent par exemple, la totalité du fragment des requêtes XPath positives de premier-ordre [12].

B.2.2 Provenance probabiliste : projection booléenne de la requête

Une requête booléenne sur un document déterministe retourne vrai ou faux. Une requête booléenne Q sur un p-documents $\mathscr{P}$ retourne la *probabilité* que Q est vraie dans $\mathscr{P}$, soit la somme des probabilités de tous les documents générés depuis $\mathscr{P}$ où Q est vraie. Une observation fondamentale [50], en raison de la nature localement monotone du langage de requête, dévoile que la probabilité d'une requête sur un p-document est la probabilité que l'une des réponses de la requête sur le document déterministe correspondant résultent aussi sur $\mathscr{P}$. Nous utilisons cette observation pour transformer notre problème d'interrogation d'XML probabiliste en un problème de calcul de probabilité sur une formule propositionnelle sous forme normale disjonctive (FND).

Reprenons l'exemple de requête Q_1 qui cherche la liste des contributeurs à la révision de l'article intitulé « Roger Waters ». En observant la figure B.5, nous pouvons voir trois instances correspondant à Q_1 sur le document déterministe qui sous-tend le p-document, correspondant aux trois différents nœuds « contributor ». Pour chaque match, nous pouvons construire la conjonction de tous les littéraux probabilistes sur le chemin à partir de la racine de l'un des nœuds appariés par la

requête :

$$c_1 = e_5 \wedge e_1 \wedge e_2$$

$$c_2 = e_5 \wedge e_2 \wedge e_3$$

$$c_3 = e_5 \wedge e_1 \wedge e_4$$

Ces conjonctions sont appelées *provenance probabiliste* de chaque match/réponse à la requête.

Comme indiqué précédemment, la probabilité d'une requête Q localement monotone sur un p-document est exactement égale à la probabilité de la disjonction des provenances probabilistes de tous les matches de la requête sur le document déterministe sous-jacent. La probabilité de Q_1 sur le p-document de La figure B.5 est donc la probabilité de la formule propositionnelle φ_1 suivante, appelée *provenance probabiliste* de la requête booléenne Q_1 sur ce document :

$$\varphi_1 = c_1 \vee c_2 \vee c_3 = (e_5 \wedge e_1 \wedge e_2) \vee (e_5 \wedge e_2 \wedge e_3) \vee (e_5 \wedge e_1 \wedge e_4).$$

Parce que les conditions de probabilité des nœuds de distribution sont composées de conjonctions de littéraux dans le modèle PrXML^{cie}, les provenances probabilistes sont toujours en forme normale disjonctive (FND). Dans ce qui suit, nous allons souvent noter cette FND $\varphi = \bigvee_{i=1}^{m} c_i$ et appelons chaque c_i une *clause* de la FND. Dans ProApproX, les événements de probabilité pour un nœud de distribution donné sont stockés dans un nœud attribut d'un document XML régulier et géré par un système de gestion natif de bases de données XML ordinaires. Le nom de cet attribut peut être spécifié par l'utilisateur ; pour les bases de données utilisées dans les expériences de la présente thèse, l'attribut a été nommé « event ».

Pour récupérer la provenance probabiliste d'une requête, nous devons transformer notre requête XPath initiale Q en une requête XQuery Q' qui retourne la concaténation de tous les valeurs d'attributs « event » de chaque match/réponse. Dans notre exemple, Q_1 se transforme en requête Q'_1 :

```
for $a in //article
for $b in $a/title/text()[.='Roger Waters']
for $c in $a//contributor
let $leaves:=($b,$c)
let $atts:=(for $i in $leaves
  return $i/ancestor-or-self::*/@event)
return <match> {
  distinct-values(
    for $att in $atts
    return string($att))} </match>
```

Un traducteur générique est implémenté dans ProApproX assurant la transformation d'une requête tree-pattern (possiblement avec jointure) exprimée en XPath une requête d'extraction de la provenance probabiliste exprimée en langage XQuery. Nous avons ensuite séparé le problème d'interrogation des bases XML probabilistes en deux problèmes indépendants :

- l'évaluation d'une requête XQuery sur un document XML ordinaire ;
- le calcul de la probabilité d'une formule propositionnelle en FND.

Le premier problème peut être résolu en utilisant n'importe quel processeur XQuery, nous utilisons BaseX, le SGBD XML natif[1] après avoir comparé ses performances avec d'autres moteurs XQuery (chapitre 5). Le second problème est le sujet du chapitre4.

B.2.3 Complexité de l'évaluation des requêtes : énoncé de la problématique

Il est facile de voir que toute formule FND peut être générée comme la provenance probabiliste d'une requête XPath même triviale comme le cas de `//A` [34]. La probabilité p d'une formule propositionnelle φ en FND dénote la probabilité qu'une assignation aléatoire de valeur vérité aux variables (ou événements) intervenant dans la FND satisfait la formule φ. S'il y a n variables booléennes, il y a 2^n assignations possibles.

Le problème de compter le nombre exact de solutions à une FND ou l'évaluation de sa probabilité est connu pour être #P-dur en général, même si tous les événements ont la même probabilité [56]. Cela implique en particulier qu'il n'y a pas d'algorithme polynomial pour l'exacte solution si $P \neq NP$. Notons que #P est une classe de complexité irréductible qui contient NP [56].

Dans le contexte d'un p-document PrXML^{cie}, cette énumération d'assignations est conceptuellement équivalente à une évaluation des requêtes sur chaque monde possible ou document ordinaire issue du p-document suivant la distribution des événements. La somme des probabilités des mondes possibles $\{\mathscr{P}_1, ..., \mathscr{P}_n\}$ qui satisfont la requête (i.e., une clause de la provenance probabiliste en FND est satisfaite), est la valeur de probabilité qui répond au problème posé.

Toutefois, l'évaluation de la probabilité d'une FND est généralement résolue via un schéma d'approximation, surtout quand une estimation de cette quantité est suffisante en pratique. Par exemple, une approximation additive à la suite d'un échantillonnage Monte Carlo peut être très efficace et conduit à une bonne précision. Néanmoins, le Monte Carlo naïf est mauvais pour le cas où le nombre d'affectations satisfaisantes pour une FND est exponentiellement petit, par rapport

[1] `http://basex.org/`

au (grand) nombre d'affectations possibles. D'un point de vue statistique, nous aurions besoin d'un nombre de tirages exponentiellement grand depuis l'ensemble des assignations possibles pour obtenir un nombre de succès recherché.

Au lieu de tirer uniformément au hasard et tester chaque assignation tirée, nous pouvons choisir de tirer seulement sur l'ensemble des assignations qui satisfont la FND (mais pas uniformément). De nombreuses stratégies *échantillonner uniquement l'espace important* ont été introduites, parmi lesquelle nous citons les algorithmes de Karp, Luby, et de Madras dans [32]. Aussi, lorsque le nombre de littéraux par clause est constant, Ajtai et Wigderson [7] proposent une solution d'approximation $(\varepsilon, \delta = 0)$ à coût linéaire en la taille de la FND et paramètres de l'approximation. Étant donnée une erreur tolérée ε sur le résultat, et un facteur de fiabilité δ pour le rapprochement, on trouve dans la littérature un certain nombre d'algorithme implémentant des schémas d'approximation polynomiaux entièrement randomisés ou FPRAS, pour le problème de l'estimation de probabilité d'une FND. Néanmoins, un FPRAS aura toujours un coût linéaire en la taille de la FND, l'erreur $(1/\varepsilon)$, et le facteur de fiabilité $(\log /1\delta)$. Ce fait laisse le problème ouvert pour les grandes FND, qui peuvent en plus contenir des clauses très corrélées, ce qui ajouterait un certain degré de difficulté au problème.

Puisque la question est fondamentalement difficile, nous visons à concevoir des solutions optimisées pour répondre à la requête de manière exacte ou approximative. L'idée repose sur une approximation de la valeur de probabilité d'une FND dure, après avoir effectué un certain nombre de simplifications. Ces opérations donnent suite à une décomposition de la provenance probabiliste en FND initiale en plusieurs sou-formules indépendantes. Nous introduisons ensuite un certain nombre d'options pour évaluer la probabilité de la décomposition, ce qui augmente alors la chance de cibler le plan optimal d'évaluation de cette formule. Nous discutons dans le reste de notre thèse d'un plan d'optimisation de calcul des probabilités, via une approche prospective qui s'appuie sur l'estimation des coûts des différents plans de calcul possibles.

Conclusions

Nous avons introduit une approche originale d'optimisation de requête comme méthode d'évaluation des résultats de requêtes sur les bases XML probabilistes. Bien que certains composants soient spécifiques à la méthode présentée (en particulier, la traduction des requêtes XPath en requêtes d'extraction de provenance probabiliste en XQuery), le noyau de notre système vise à résoudre le problème très général de l'évaluation (approximative) et efficace de la probabilité d'une formule propositionnelle se composant de variables aléatoires indépendantes. Nos techniques de décomposition de la provenance probabiliste ainsi que les algorithmes utilisés

supposent que la formule est en FND, mais la technique pourrait probablement être étendue à des formules arbitraires.

Le premier principe sur lequel repose notre approche, ainsi que les principales observations concluses des expériences, c'est que *l'algorithme optimal pour l'évaluation de la probabilité à utiliser dépend fortement des caractéristiques de la formule* : si la formule a peu de variables, utiliser l'algorithme naïf ; si elle est composée de peu de clauses, opter pour l'algorithme de sieve (inclusion-exclusion), si la probabilité est connue pour être proche de 1, utiliser le Monte-Carlo naïf, si la formule est obtenue à partir de l'évaluation d'une requête en tree-pattern sur une structure en forme arborescente du modèle e dépendance locale, utiliser une technique telle que EvalDP, si rien d'autre ne fonctionne, utilisez l'algorithme Self-Adjusting Coverage. Notre modèle de coûts tient ceci en compte. Le deuxième principe est que *différents algorithmes peuvent être utilisés pour évaluer la probabilité de différentes parties d'une formule*. Notre technique de décomposition de la formule, et l'exploration des plans possibles d'évaluation, font usage de cette constatation.

Dans ProApproX, l'extraction de la provenance probabiliste et l'évaluation de la probabilité de cette dernière constituent deux problèmes différents. Chacun de ces problèmes peut ainsi être optimisé indépendamment (en utilisant, respectivement, l'optimisation des requêtes XML effectuée par un SGBD XML natif, et notre technique d'évaluation de probabilité d'une formule propositionnelle en FND). Cependant, il est probable que l'exploitation de la structure de la requête pour obtenir une provenance probabiliste déjà factorisée entraînerait une évaluation encore plus efficace. Il s'agit d'une direction pertinente pour des recherches futures.

Nous concluons cette thèse en mettant en évidence d'autres optimisations intéressantes et possibles sur la solution discutée, et un certain nombre de perspectives au sujet de l'interrogation des données XML probabilistes. Commençons par les optimisations :

- Il pourrait être utile d'affiner notre modèle de coût avec des fonctions non linéaires de la taille des formules, afin de tenir en compte les termes de second-ordre dans la complexité des algorithmes ;

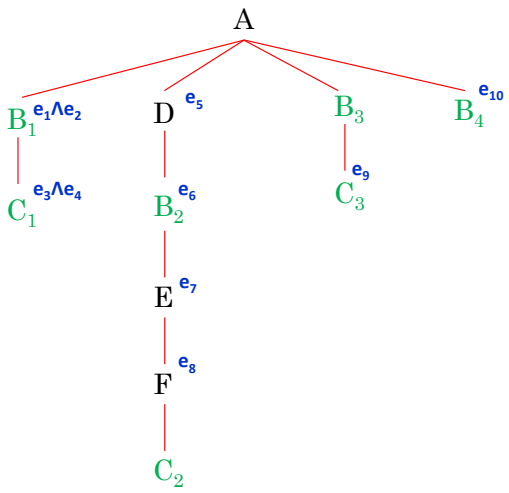
- Il ya aussi la question d'une exploration plus intelligente de l'espace des plans d'évaluations possibles, en considérant l'attribution de différentes valeurs à ε et δ aux nœuds approximés. Bien qu'il soit toujours préférable d'appliquer l'algorithme naïf sur deux enfants d'un nœud que sur le nœud lui-même, ce constat n'est pas vrai pour les algorithmes d'approximation en général, cela nécessiterait donc une analyse non triviale des différents cas ;

- On peut également effectuer l'expérience de garder la même valeur du paramètre δ à travers tout l'arbre en effectuant des tirages corrélés dans

toutes les branches de l'arbre – ceci n'est pas trivial également, puisque nous aurons besoin de garder une trace de tous les tirages partiels exécutés.

Problèmes ouverts

Requêtes avec négation Une direction ultérieure à suivre est le support des requêtes négatives. Cette nouvelle classe de requêtes peut nécessiter des efforts, mais contient un grand nombre de cas faciles, avec peut-être quelques difficultés inhérentes pour d'autres. En effet, l'extension de notre solution afin de traiter cette classe de requêtes est possible pour les requêtes dont la provenance probabiliste est en forme normale disjonctive. Notez cependant que, généralement, les requêtes négatives sont susceptibles de retourner une FND exponentiellement grande en la taille de la base de données. Le défi technique pour les requêtes négatives est l'extraction de la provenance probabiliste, qui devrait suivre une interprétation sémantique particulière sur l'arbre. Prenons l'arbre PrXML suivant :



Lorsque nous cherchons un nœud B qui ne dispose pas d'un nœud C descendant, le système doit aussi inclure un chemin donné de B à un nœud probabiliste C, comme ce dernier a une probabilité de ne pas apparaître. La requête négative simple en tree-pattern Q `://B[not(.//C)]` sur l'arbre précédent aura alors cinq matches possibles, chacune avec sa provenance probabiliste respective qui manifeste la négation :
$c_1 = e_1 \wedge e_2 \wedge \neg(e_3 \wedge e_4) = (e_1 \wedge e_2 \wedge \neg e_3) \vee (e_1 \wedge e_2 \wedge \neg e_4)$: ce chemin probabiliste exprime la condition de l'apparition de B_1 et non-apparition de C_1 ;
$c_2 = (e_5 \wedge e_6 \wedge \neg e_7)$: apparition de D et B_2, et la non-occurrence de E ;
$c_3 = (e_5 \wedge e_6 \wedge e_7 \wedge \neg e_8)$: apparition de D et B_2, occurrence de E, et non-occurrence de F ;

$c_4 = (\neg e_9)$: la non-survenance de C_3 ;
$c_5 = (e_{10})$: apparition de B_4.

Parfois, le développement des clauses résulte en une provenance équivalente en FND, comme dans le cas de la provenance de la requête exemple Q. Dans tous les cas, nous aurons besoin de concevoir un nouveau mécanisme d'extraction de la provenance probabiliste des requêtes. Notons également que pour le cas où une requête comporte plus d'un prédicat de négation, de nouveaux aspects doivent être pris en considération lors de l'extraction de sa provenance.

Requêtes d'agrégation Avec l'émergence des données probabilistes dans de nombreux domaines d'application importants, le besoin exprimé par la communauté scientifique et autres utilisateurs de comprendre et de traiter efficacement les requêtes d'agrégation est prévu d'augmenter dans un futur proche. Le résultat d'une requête qui fait usage de ces fonctions d'agrégation est un ensemble de valeurs possibles (pour chaque document possible), chacune avec sa probabilité. Une étude théorique qui examine les propriétés sous-jacentes liées à la sémantique riches de requêtes de classement et d'agrégation de données probabilistes, est donnée dans [3], qui introduit également des distributions continues (Par exemple, les données fournies par les capteurs [14, 20]). Une sémantique de la provenance probabiliste des requêtes d'agrégation dans le schéma relationnel est donnée dans [8], alors que le problème de l'évaluation de sa probabilité est étudiée par R. Fink et al. dans [21].

Distribution du calcul ProApproX montre déjà de bonnes performances sur les données de grande taille, comme démontré de façon empirique à travers les expériences dur les données synthétiques. Mais il est possible de recourir à un calcul distribué, dans le cas où, par exemple, un seuil sur la taille d'une formule en FND entrée est atteint. En cas d'un très grand nombre de clauses, il est possible d'appliquer l'opérateur *ou-indépendant* pour tenter de diviser la FND sur multi-cœur ou dans une architecture distribuée. Les différentes valeurs calculées seront ensuite agrégées par une disjonction indépendante pour former le résultat de la requête. Il est également possible d'exploiter le fait que la plupart des algorithmes d'approximation gardent une efficacité de performance en passant à l'échelle.