Storing and Querying Probabilistic XML Using a Probabilistic Relational DBMS

E.S. Hollander and M. van Keulen

Faculty of EEMCS, University of Twente, Enschede, The Netherlands
{e.s.hollander;m.vankeulen}@ewi.utwente.nl

Abstract. This work explores the feasibility of storing and querying probabilistic XML in a probabilistic relational database. Our approach is to adapt known techniques for mapping XML to relational data such that the possible worlds are preserved. We show that this approach can work for any XML-to-relational technique by adapting a representative schema-based (inlining) as well as a representative schemaless technique (XPath Accelerator). We investigate the maturity of probabilistic relational databases for this task with experiments with one of the state-of-the-art systems, called Trio.

1 Introduction

Data in a database is typically treated as being correct and indisputable. In many applications, this obviously is not really true. For example, data may be out of date, or some value may just be the most likely one and could very well be wrong. This is even more true for the results of automatic tasks like information extraction, natural-language processing, data integration, sensor data management, or data mining. To better support such applications, there is growing interest in the management of *uncertain data*, i.e., data for which we explicitly store the fact that it is uncertain together with information about its uncertainty.

In many of these applications, data is semi-structured, because a hierarchical representation is natural or when a source is already in this form [1]. Most research in the database community, however, is directed towards probabilistic relational databases. Several research prototype systems have been released into the open source community such as MayBMS [2, 3], Trio [4, 5], Mystiq [6], and Orion [7]. Although receiving less attention, uncertain semi-structured data, and in particular probabilistic XML, has also been used as a data model for uncertain data [8–10]. As far as we know, the work of Kimelfeld et al. is the only truly in depth work on querying probabilistic XML [11].

The contribution of this paper is twofold: (1) We present an approach for adapting existing XML-to-relational mapping techniques that preserves the possible worlds. We show how to concretely accomplish this by adapting a representative schema-based (inlining) [12] as well as a representative schemaless technique (XPath Accelerator) [13]. These lie, for example, at the heart of Oracle's object-relational storage schema [14] and MonetDB/XQuery [15], respectively.

Kind	Description
ind	*Independent choice* for each of its children.
mux	*Mutual exclusive choice* for one of its children.
det	*Deterministic choice* of all of its children. Often used in combination with ind or mux to choose multiple children in an all-or-nothing manner.
exp	*Explicit choice* of certain specific subsets of children.
cie	A choice based on a *conjunction of independent events*.

Table 1. Kinds of distributional nodes [1]

(2) Furthermore, we investigate the maturity of probabilistic relational databases for this application by experimenting with a few queries on mapped data of some XML documents on one of the state-of-the-art probabilistic database systems, namely Trio. Note that although we illustrate the adaptation of mapping techniques also with Trio, it is fairly straightforward to transfer the approach to the data models of other probabilistic databases.

2 Probabilistic Databases

2.1 Possible world theory

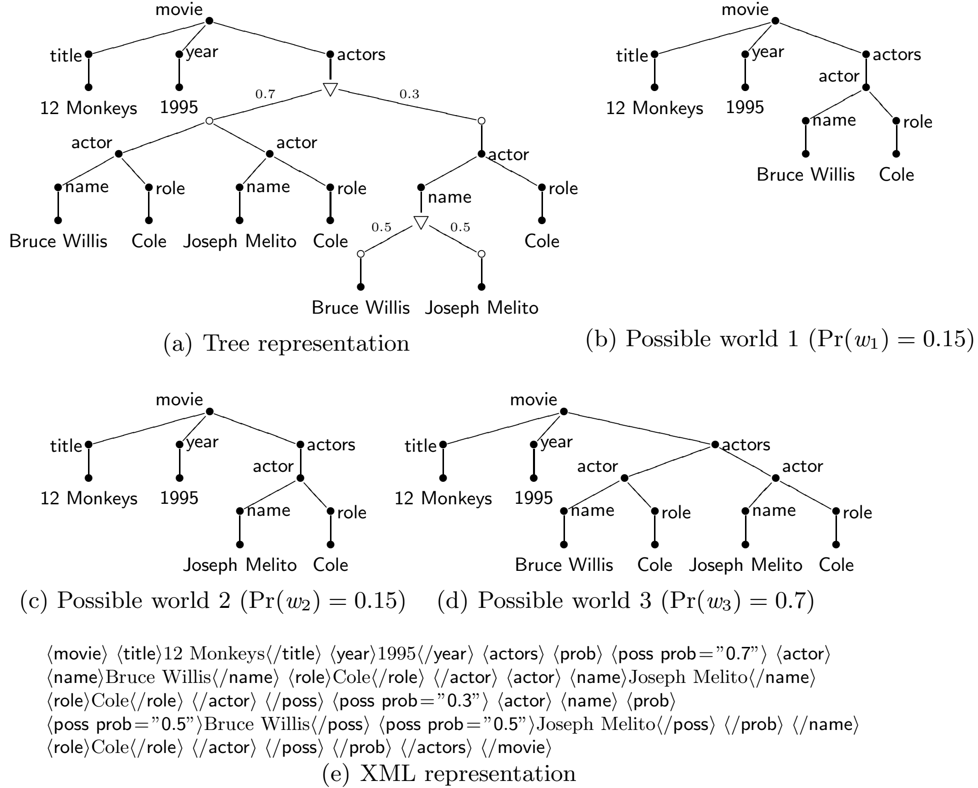
A probabilistic database PDB is a set of possible worlds $PDB = \{w_1, \ldots, w_n\}$ each with its probability $\Pr(w_i)$ such that $\sum_{i=1..n} \Pr(w_i) = 1$. Each world is an ordinary database, so in case of probabilistic XML, a world is an ordinary XML tree and in case of probabilistic relations, a world is a set of ordinary relations.

The semantics of a query on a probabilistic database are defined as the set of answers of the query posed to each of the possible worlds individually. Consequently, the probability of a particular answer is the sum of the probabilities of all possible worlds for which the query produced that answer.

Since the number of possible worlds grows exponentially, implementations of probabilistic databases store all possible worlds in one compact representation. Algorithms for querying a probabilistic database directly work on the compact representation while strictly adhering to the semantics of querying as defined in terms of possible worlds.

2.2 Probabilistic XML

Probabilistic XML is an extension to XML for representing uncertainty in the data in a compact way. This is achieved by introducing *distributional nodes* to denote probabilistic distributions over subtrees (see Tab.1). There are several families of probabilistic XML with varying expressiveness depending on the kinds of distributional nodes allowed [1]. In this paper, we use the probabilistic XML model of [10, 16] which is equivalent with the $\mathsf{PrXML}^{\{\mathsf{mux},\mathsf{det}\}}$ family. In this model, mux nodes are called *probability nodes* (denoted with ⟨prob⟩ in XML and



(a) Tree representation

(b) Possible world 1 ($\Pr(w_1) = 0.15$)

(c) Possible world 2 ($\Pr(w_2) = 0.15$)

(d) Possible world 3 ($\Pr(w_3) = 0.7$)

(e) XML representation

Fig. 1. Example of Probabilistic XML.

$\bigtriangledown$ in the tree representation) and det nodes are called *possibility nodes* (denoted with ⟨poss⟩ in XML and $\circ$ in the tree representation).

Example 1. Figure 1 shows an example of a probabilistic XML instance. It is uncertain if there are two actors playing the role "Cole", or that there is just one actor playing the role, but in this case it is uncertain which of the two names is the name of the actor. Figures 1(a), 1(b)–1(d), and 1(e) respectively illustrate the tree representation, the three possible worlds it encodes, and the XML representation. Note that this is a nested model, hence a possibility node may contain an entire subtree which may in turn contain distributional nodes.

2.3 Probabilistic Relations

In recent years, there has been much research into probabilistic relational databases culminating into several prototypes such as MayBMS [2] and Trio [4]. We have used Trio in this paper for our experiments.

Trio allows for multiple alternatives for a tuple. A tuple containing more than one alternative is called an x-tuple. Alternatives may or may not have associated probabilities (called confidence scores). If the probabilities do not add up to 1, the x-tuple is called a maybe x-tuple, because it is also possible that it does not exist at all. Figure 2 shows an example of an uncertain table in Trio with one x-tuple with two alternatives $t_{1,1}$ and $t_{1,2}$. Trio is also based on possible worlds theory, hence the example encodes two possible relations.

movie

	id	**title**	**year**	
$t_{1,1}$	1	Twelve Monkeys	1995	0.7
$t_{1,2}$	1	12 Monkeys	1995	0.3

Fig. 2. Example Trio table

In principle, x-tuples are independent, i.e., arbitrary combinations of alternatives from different x-tuples can exist and they do so with a probability that is the product of the probabilities of the original alternatives. To be able to express dependencies, Trio supports *lineage*. An alternative's existence in this way may depend on the existence of other alternatives. Lineage is expressed with a boolean formula such as $\lambda(t_{1,1}) = t_{2,1} \wedge t_{3,2}$. The set of possible worlds is restricted to those where the lineage formulas are true, in our example, to those where $t_{1,2}$, $t_{2,1}$, and $t_{3,2}$ co-exist. Lineage is typically introduced by queries, because the alternatives in the result depend on the alternatives in the base tables.

3 Storing and Querying XML in Relational Databases

In recent years, many approaches to storing and querying XML have been proposed. The ones mapping XML onto relational tables can be divided into two categories: schema-based and schemaless. The former constructs a relational schema based on the DTD or XML schema of the XML documents. The latter treats the XML documents as trees and stores each node of the tree as a tuple in one generic relation. We took two techniques representative for each category: Inlining and XPath Accelerator, respectively. We summarize both below. For details, we refer to [12] and [13, 15, 17], respectively.

```
movie: (title, year, actors)
actors: (actor*)
actor: (name, role)
title: (PCDATA)
year: (PCDATA)
name: (PCDATA)
role: (PCDATA)
```

Fig. 3. Example DTD

3.1 Schema-based: Inlining

The inlining technique by Shanmugasundaram et al. [12] was one of the first algorithms available that could store an XML-document in a relational database. It relies on DTDs to generate a relational schema. The technique first constructs a DTD graph in which each node represents an element type; the arrows are annotated with the multiplicities. In general, each element type generates one table; the graph is used, however, to *inline* the information of certain elements into the table of its parent in an attempt to reduce the number of tables.

Example 2. For example, suppose we have the DTD of Fig.3. The hybrid inlining technique recognizes that 'title', 'year', 'name', and 'role' can only occur once

in their respective parent elements. Therefore, they are inlined to produce the following relations:

- movie(id:int, title:string, year:int, actorsid:int)[1]
- actor(id:int, parent:int, name:string, role:string)

These two relations suffice to store all data that is contained in XML documents conforming to this DTD.

3.2 Schemaless: XPath Accelerator

The XPath Accelerator [15] technique does not depend on the existence of a schema. Instead, it views the XML document as a tree and uses one table that stores both the information contained in the nodes as well as the structure of the tree. In this paper, we use a simplified version of the XPath Accelerator version described in [17]. The structure of the tree is encoded using three attributes:

- **pre**: the rank assigned to the node in a preorder traversal of the tree.[2]
- **size**: the size of the subtree below the node.
- **level**: the depth of the node in the tree, i.e, the length of the path from the node to the root.

pre	**size**	**level**	**kind**	**prop**
1	11	1	elem	movie
2	2	2	elem	title
3	1	3	text	Twelve Monkeys
4	2	2	elem	year
5	1	3	text	1995
6	6	2	elem	actors
7	5	3	elem	actor
8	2	4	elem	name
9	1	5	text	Bruce Willis
10	2	4	elem	role
11	1	5	text	Cole

Fig. 4. XPath Accelerator example

We use two additional attributes, kind and prop. The former contains the node kind. The latter contains for elements its tag and for text nodes its string content. Without loss of generality, we restrict ourselves to only element and text nodes. Storing, for example, the possible world of Fig.1(b) in this manner produces the table of Fig.4.

4 Mapping Probabilistic XML to Probabilistic Relations

Naively applying the techniques of Sec.3 for storing and querying XML using a relational database defies our purpose. If we would do that, we would end up with a certain database where all probability and possibility nodes are stored directly. Instead, we would like to leverage the functionality of the probabilistic RDBMS for storing and querying uncertain data. We therefore adapt the techniques in such a way that we *can* leverage this functionality.

[1] There are variants of the inlining techique that would also produce an 'actors' relation. Since it is superfluous here, we inline it for simplicity of the running example.

[2] "In a preorder traversal, a tree node v is visited and assigned its preorder rank $pre(v)$ before its children are recursively traversed from left to right." [13]

4.1 General approach

Both probabilistic XML and probabilistic relations are based on possible worlds theory. Therefore, the semantics of a query are defined in the same way: as the set of the answers to the query for each possible world. To be able to leverage the functionality of probabilistic relational database, we need to make sure that the stored relational data encodes the same possible worlds as the original probabilistic XML.

The key to preserving the possible worlds lies in the observation that uncertain data is all about choices. In probabilistic XML we choose among subtrees, in probabilistic relations we choose among alternative tuples. In the sequel, we first view both models more formally in terms of choices and then show how to adapt the XML-to-relational techniques so that they preserve the possible worlds.

4.2 Viewing Probabilistic XML in terms of choices

Probability nodes (as all distributional nodes) can be seen as independent random variables. Their domains consist of (references to) their possibility node children. Let n_1 and n_2 be the higher and lower probability nodes in Fig. 1(a), respectively, and x_{n_i} its associated random variable. The assignment $x_{n_2} \leftarrow 1$ denotes the choice for the first (left) subtree of n_2, i.e., the name "Bruce Willis", and $x_{n_2} \leftarrow 2$ denotes the choice for the second (right) subtree, i.e., the name "Joseph Melito". The probability of an assignment $\Pr(x_{n_2} \leftarrow j)$ is the probability associated with possibility node j below n_2.

A complete choice θ is a set containing one assignment for each random variable. Each complete choice determines one particular possible world w_θ with probability $\prod_{(x \leftarrow j) \in \theta} \Pr(x \leftarrow j)$. Note that because probability nodes may be nested, it may happen that two different complete choices determine the same possible world (e.g., $\{x_{n_1} \leftarrow 1, x_{n_2} \leftarrow 1\}$ and $\{x_{n_1} \leftarrow 1, x_{n_2} \leftarrow 2\}$ denote the same possible world, namely Fig.1(d)).

Viewing it from an XML node's perspective, the node only exists if it has been chosen, i.e., it only exists in those worlds determined by a complete choice that includes for each of its parent probability nodes n, the assignment $x_n \leftarrow j$ where j is a reference to the possibility node child of n that is a parent of the XML node.

4.3 Viewing probabilistic relations in terms of choices

In probabilistic relations, each x-tuple can be seen as a random variable x. Its domain consists of (references to) its alternatives. If it is possible that the tuple does not exist at all, there is a special value '$\perp$' in the domain. The remaining probability mass is given to $x \leftarrow \perp$. Here too a complete choice determines a possible world, hence the probabilistic relation of Fig.2 encodes two possible worlds, i.e., two possible movie relations.

The lineage of Trio restricts the set of possible worlds to valid ones, i.e., to only those that respect the co-existence relationships between alternatives as defined by the lineage. In terms of random variables, only those complete choices are valid for which its assignments respect the lineage, i.e., if an assignment is associated with an alternative that needs to co-exist with other alternatives, then their associated assignments are also contained in the complete choice.

Other probabilistic databases have different data models and means to restrict the set of possible worlds. MayBMS, for example, associates a set of random variable assignments (called world set descriptor) with each tuple. Therefore, it is fairly straightforward to transfer our approach to other probabilistic databases.

5 Schema-based mapping: Adapted Inlining

Our adapted inlining technique has three phases.

(1) We first construct an *event table*, i.e., an uncertain table with all random variables (attribute rvar) and their possible assignments (attribute ass).

(2) We then map the data in the XML nodes to certain relational tables in the same way as the inlining technique prescibes except that we do not inline child element types that may contain uncertainty (see Sec.5.3). We furthermore mark the tuples with the ids of the event associated with its direct parent possibility node (it is not necessary to also mark them with the ids of the other ancestor possibility nodes as we will see later). XML nodes that do not have a parent possibility node (certain XML nodes) are marked with `NULLs`.

(3) Finally, we execute queries that produce the same tables, but with the proper lineage attached expressing the dependence on the various random variable assignments. For 'movie' we execute the following query:

```
CREATE TABLE umovie AS
   SELECT movie.id, movie.title, movie.year, movie.actorsid
   FROM   movie, event
   WHERE (movie.rvar = event.rvar AND movie.ass = event.ass)
   OR    movie.rvar IS NULL;
```

5.1 Nesting

If the probabilistic XML document contains nested elements, step 3 above is performed from top to bottom. This happens in our example for 'actor' which is a decendant of 'movie'. Since one movie can have multiple actors, the inlining technique creates another table 'actor' which contains an attribute with a reference to the parent. There may exist uncertainty surrounding the actors as well. In a probabilistic XML tree, a node can only exist when its parent node also exists. These dependencies need to be stored correctly.

This is where lineage fully comes into play. Trio ensures that the existence of a tuple depends on the existence of the tuples in its lineage (and the lineage thereof, and so on). Therefore, we create the uncertain tables for child element types

event

	rvar	ass	
$t_{1,1}$	1	1	0.7
$t_{1,2}$	1	2	0.3
$t_{2,1}$	2	1	0.5
$t_{2,2}$	2	2	0.5

movie

	rvar	ass	id	title	year	actorsid
t_3	NULL	NULL	1	12 Monkeys	1995	1

umovie

	id	title	year	actorsid	
t_4	1	12 Monkeys	1995	1	$\lambda(t_4) = t_3$

actor

	rvar	ass	id	parent	role
t_5	1	1	1	1	Cole
t_6	1	1	2	1	Cole
t_7	1	2	3	1	Cole

uactor

	id	parent	role	
t_8	1	1	Cole	$\lambda(t_8) = t_{1,1} \wedge t_5 \wedge t_4$
t_9	2	1	Cole	$\lambda(t_9) = t_{1,1} \wedge t_6 \wedge t_4$
t_{10}	3	1	Cole	$\lambda(t_{10}) = t_{1,2} \wedge t_7 \wedge t_4$

name

	rvar	ass	id	parent	text
t_{11}	NULL	NULL	1	1	Bruce Willis
t_{12}	NULL	NULL	2	2	Joseph Melito
t_{13}	2	1	3	3	Bruce Willis
t_{14}	2	2	4	3	Joseph Melito

uname

	id	parent	text	
t_{15}	1	1	Bruce Willis	$\lambda(t_{15}) = t_{11} \wedge t_8$
t_{16}	2	2	Joseph Melito	$\lambda(t_{16}) = t_{12} \wedge t_9$
t_{17}	3	3	Bruce Willis	$\lambda(t_{17}) = t_{2,1} \wedge t_{13} \wedge t_{10}$
t_{18}	4	3	Joseph Melito	$\lambda(t_{18}) = t_{2,2} \wedge t_{14} \wedge t_{10}$

Fig. 5. Inlining-based mapping of example XML.

based on the resulting uncertain tables of their parent element types created previously. This ensures that the lineage expresses all dependencies in the tree.

We can create the table 'uactor' by issuing the following query ('uname' analogously).

```
CREATE TABLE uactor AS
SELECT actor.id, actor.parent, actor.role
FROM   actor, event, umovie
WHERE  ((actor.rvar = event.rvar AND actor.ass = event.ass)
OR      actor.event IS NULL)
AND    actor.parent = umovie.id;
```

5.2 Example

Figure 5 shows the result for the example tree of Fig.1(a). Note that we invented new identifiers for the tuples. Also note that 'uactor' and 'uname' contain only x-tuples with one alternative instead of more as you might expect. The lineage, however, expresses that these x-tuples depend on $t_{1,1}$, $t_{1,2}$, $t_{2,1}$, and $t_{2,2}$. Since the first two are mutually exclusive and last two as well, the lineage dependencies make some of the other x-tuples to be mutually exclusive as well.

Suppose we have a possible world in which $t_{1,1}$ exists. This corresponds to Fig. 1(d). In this possible world, t_{10} cannot exist, because it depends on $t_{1,2}$, which is mutually exclusive with $t_{1,1}$. Further down the tree, t_{17} cannot exist

either, because it depends on t_{10}. In this way, lineage preserves the dependencies that exist in the original tree.

We used a cartesian product of both tables, hence the event table must not be empty. This could happen if the probabilistic XML tree is certain. This problem can be avoided by using an outer join instead of the cartesian product. Unfortunately, this functionality was not available in Trio at the time of writing.

Since we refer to the parent table that already contains lineage, these will be taken into account when querying. In this way, we fully leverage Trio's functionality for calculating probabilities in the context of complex dependencies among x-tuples. Finally, observe that the resulting probabilistic relations encode the same possible worlds as the original probabilistic XML document.

5.3 Avoiding data duplication

Data duplication may occur if we would inline a value that is uncertain, because if an x-tuple contains an inlined uncertain attribute, it may result in several alternatives. In each of these alternatives, the other certain attributes are duplicated. In our example, this happens with 'name' and 'role': both 'name' and 'role' could be inlined according to the original inlining technique, but 'name' is uncertain, therefore if we would inline 'name' as well, the certain data (in our example the value "Cole") is duplicated in the alternatives for the name.

If more than one inlined attribute is uncertain, data duplication would grow exponentially. For example, if 'role' would be uncertain as well with two alternatives, then we end up with 4 alternatives for the one actor x-tuple.

The solution to this problem is to not inline element types which may be uncertain. As a consequence, the element type gets a relation of its own, with an accompanying reference to the parent tuple. Note that, in this way, values occur as many times as in the orginal document.

6 Schemaless mapping: Adapted XPath Accelerator

For XPath Accelerator, we calculate the pre, size and level values for every node in the probabilistic XML document, i.e., including the distributional nodes. We store the data for the XML nodes in the 'doc' table and the data for the possibility nodes in the event table where all possibility nodes of one probability node form one x-tuple. We also add a 'catch all' event t_0 (otherwise we would not select any certain nodes at the root of the document).

We combine the two tables in the same way as for the inlining technique except for the fact that we do not have ids for relating tuples in doc with the x-tuples in event. Instead, each node in the probabilistic XML document depends on all its ancestor possibility nodes. The ancestor-relationship can be expressed using the pre and size attributes. Hence we execute the following query:

event

	pre	size	level	
t_0	0	29	0	1
$t_{1,1}$	8	11	4	0.7
$t_{1,2}$	19	9	4	0.3
$t_{2,1}$	23	2	8	0.5
$t_{2,2}$	25	2	8	0.5

doc

	pre	size	level	kind	prop
t_3	1	28	1	elem	movie
t_4	2	2	2	elem	title
t_5	3	1	3	text	12 Monkeys
t_6	4	2	2	elem	year
t_7	5	1	3	text	1995
t_8	9	5	5	elem	actor
t_9	10	2	6	elem	name
	⋮	⋮	⋮	⋮	⋮
t_{10}	20	9	5	elem	actor
t_{11}	21	2	6	elem	name
t_{12}	24	1	9	text	Bruce Willis
t_{13}	26	1	9	text	Joseph Melito
t_{14}	27	2	6	elem	role
t_{15}	28	1	7	text	Cole

udoc

	pre	size	level	kind	prop	
t_{16}	1	28	1	elem	movie	$\lambda(t_{16}) = t_0 \wedge t_3$
t_{17}	2	2	2	elem	title	$\lambda(t_{17}) = t_0 \wedge t_4$
t_{18}	3	1	3	text	12 Monkeys	$\lambda(t_{18}) = t_0 \wedge t_5$
t_{19}	4	2	2	elem	year	$\lambda(t_{19}) = t_0 \wedge t_6$
t_{20}	5	1	3	text	1995	$\lambda(t_{20}) = t_0 \wedge t_7$
t_{21}	9	5	5	elem	actor	$\lambda(t_{21}) = t_0 \wedge t_{1,1} \wedge t_8$
t_{22}	10	2	6	elem	name	$\lambda(t_{22}) = t_0 \wedge t_{1,1} \wedge t_9$
	⋮	⋮	⋮	⋮	⋮	
t_{23}	20	9	5	elem	actor	$\lambda(t_{23}) = t_0 \wedge t_{1,2} \wedge t_{10}$
t_{24}	21	2	6	elem	name	$\lambda(t_{24}) = t_0 \wedge t_{1,2} \wedge t_{11}$
t_{25}	24	1	9	text	Bruce Willis	$\lambda(t_{25}) = t_0 \wedge t_{1,2} \wedge t_{2,1} \wedge t_{12}$
t_{26}	26	1	9	text	Joseph Melito	$\lambda(t_{26}) = t_0 \wedge t_{1,2} \wedge t_{2,2} \wedge t_{13}$
t_{27}	27	2	6	elem	role	$\lambda(t_{27}) = t_0 \wedge t_{1,2} \wedge t_{14}$
t_{28}	28	1	7	text	Cole	$\lambda(t_{28}) = t_0 \wedge t_{1,2} \wedge t_{15}$

Fig. 6. XPath Accelerator-based mapping of example XML.

```
CREATE TABLE udoc AS
   SELECT DISTINCT doc.pre, doc.size, doc.level, doc.kind, doc.prop
   FROM   doc, event
   WHERE  doc.pre>event.pre AND doc.pre<(event.pre+event.size);
```

Unfortunately, this query produces the 'udoc' table of Fig.7 instead of the desired result of Fig.6. If an XML node depends on more than one ancestor possibility node, then the `DISTINCT` produces *or*-lineage as opposed to *and*-lineage. For example, the lineage of tuple t_8 does not contain the term $t_0 \wedge t_{1,1}$ but $t_0 \vee t_{1,1}$. At the time of writing, Trio does not have a keyword or some other construct that allows us to specify that tuples are dependent on all tuples that correspond with a certain predicate. Trio does support *and*-lineage, we only cannot construct it under the circumstances we have at hand here. To be able to reliably conduct our experiments, we have manually updated the underlying

udoc

	pre	size	level	kind	prop	
t_{24}	21	2	6	elem	name	$\lambda(t_{24}) = (t_0 \vee t_{1,2}) \wedge t_{11}$
t_{25}	24	1	9	text	Bruce Willis	$\lambda(t_{25}) = (t_0 \vee t_{1,2} \vee t_{2,1}) \wedge t_{12}$
t_{26}	26	1	9	text	Joseph Melito	$\lambda(t_{26}) = (t_0 \vee t_{1,2} \vee t_{2,2}) \wedge t_{13}$

Fig. 7. Excerpt of wrong or-lineage based udoc-result.

File	Size	#XML nodes	#▽ nodes	#○ nodes	Avg #○ per ▽
1	10.4 kB	784	1	1	1
2	43.7 kB	2510	119	240	2.016807
3	54.9 kB	3193	129	265	2.054264
4	119.7 kB	7340	193	425	2.202073
5	186.7 kB	11490	255	570	2.235294
6	800.0 kB	52320	801	1872	2.337079

Fig. 8. Data set properties

PostgreSQL tables that encode the lineage for these tables such that the resulting table is associated with the exact lineage we need.

Many XML-to-relational mapping techniques are based on prefix-labelling schemes, such as ORDPATHs and DeweyIDs. These serve both as node ID as well as efficient ways of determining axis relationships. Since these are likely to produce similar query characteristics as the pre/size conditions of the XPath Accelerator, we have not investigated these approaches seperately. Note that here too, values occur as many times as in the XML document, so space complexity is the same.

6.1 Query mapping

Our objective is to evaluate queries on the probabilistic XML data using the probabilistic relational backend. Having mapped the probabilistic XML data onto probabilistic relations according to the inlining or the XPath Accelerator technique in this way, mapping the queries is trivial: we can simply apply the same approach as in the original inlining and XPath Accelerator unaltered. Because the data represents exactly the same possible worlds, and each of the possible worlds conforms to the original techniques, the query answer conforms to the possible world semantics. See Sec. 7.2 for an example.

7 Experiments

7.1 Experimental setup

We experiment with real-life uncertain data obtained from a probabilistic data integration application [18]. The application integrates movie data from TV guide (`www.tvguide.com`) with the Internet Movie Database (`www.imdb.com`). For this

Inlining:

```
SELECT y.year
FROM umovie m, utitle t, uyear y
WHERE m.id = t.parentid
  AND t.title = 'District B13'
  AND m.id = y.parentid;
```

XPath Accelerator:

```
CREATE TABLE temp1 AS
 SELECT DISTINCT v1.*
 FROM udoc c, udoc v1
 WHERE v1.pre > c.pre AND v1.pre < (c.pre + c.size)
   AND v1.kind = 'elem' AND v1.prop = 'movie';
CREATE TABLE temp2 AS
 SELECT DISTINCT v1.*
 FROM temp1 c, udoc v1
 WHERE v1.pre > c.pre AND v1.pre < (c.pre + c.size)
   AND v1.kind = 'elem' AND v1.prop = 'title';
CREATE TABLE temp3 AS
 SELECT DISTINCT v1.*
 FROM temp2 c, udoc v1
 WHERE v1.pre > c.pre AND v1.pre < (c.pre + c.size)
   AND v1.kind = 'text' AND v1.prop = 'District B13';
CREATE TABLE temp4 AS
 SELECT DISTINCT v1.*
 FROM temp3 c, udoc v1
 WHERE v1.pre < c.pre
   AND (v1.pre + v1.size) >= (c.pre + c.size)
   AND v1.kind = 'elem' AND v1.prop = 'movie';
CREATE TABLE temp5 AS
 SELECT DISTINCT v1.*
 FROM temp4 c, udoc v1
 WHERE v1.pre > c.pre AND v1.pre < (c.pre + c.size)
   AND v1.kind = 'elem' AND v1.prop = 'year';
SELECT DISTINCT v1.*
 FROM temp5 c, udoc v1
 WHERE v1.pre > c.pre AND v1.pre < (c.pre + c.size)
   AND v1.kind = 'text';
```

Fig. 9. Translations of the exact match query.

paper, it is not important to understand much about probabilistic data integration (the reader is referred to [16]), only that it semi-automatically fuses two XML documents producing a probabilistic XML document. By varying some thresholds in the probabilistic integration, we obtain probabilistic XML documents with varying amounts of uncertainty, hence of varying sizes (see Tab.8).

We experiment with 3 queries representative for 3 categories of querying:

1. [Exact match] `//movie[title='District B13']/year/text()`
   "The year in which the movie 'District B13' has been released"
2. [Tree navigation] `//movie[actors/actor/name='Brooke Smith']/title/text()`
   "The titles of all movies in which Brooke Smith is an actor"
3. [Join] `//movie[year=//movie[actors/actor/name='David Belle']/year]/title/text()`
   "The titles of all movies of the same year as a movie with actor David Belle"

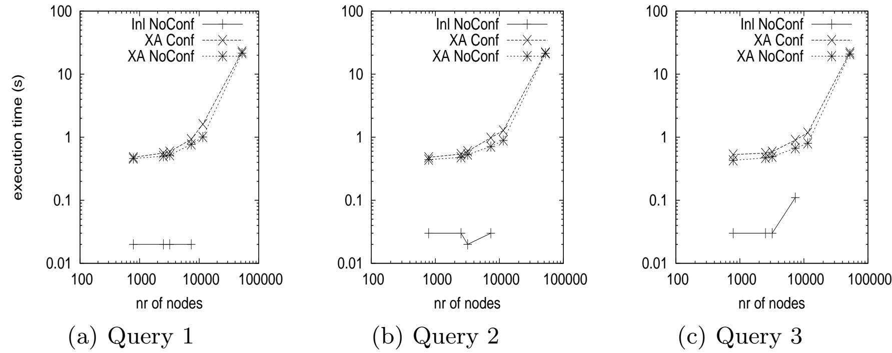
Note that movie titles, actor names and years are often ambiguous in the integration scenario, hence the chosen queries deliberately target highly uncertain sections of the resulting documents.

The experiments are performed on a PC with an AMD Athlon 64 X2 Dual Core 4000+ processor, 1 GB of internal memory and Windows XP Service Pack 3 installed. We have used PostgreSQL 8.2 as backend for Trio.

There are in total 72 runs in our experiments, namely one run for each combination of 6 data sets, 3 queries, 2 techniques, and with or without confidence calculation. For each run, we measured average query execution time for 5 executions on a hot database.

7.2 Query translation

Obviously, the abovementioned XPath queries need to be translated to SQL according to the technique involved: Inlining or XPath Accelerator. See Fig. 9 for



(a) Query 1 (b) Query 2 (c) Query 3

Fig. 10. Experimental results.

the translations of the exact match query. The translation for XPath Accelerator deviates from what is prescribed for this technique. XPath Accelerator requires self-joins for evaluating XPath steps. Unfortunately, Trio could not cope with the number of self-joins. Therefore, we split the query into one per step. We store the intermediate result in a table to be queried in the next step. This splitting also does not permit nested predicates, so we rewrote the query to the equivalent `//movie/title[.='District B13']/ancestor::movie/year/text()` before translating it.

7.3 Results

We were unable to obtain measurements for all 72 runs of the experiment due to practical problems with Trio:

- We have no measurements for Inlining queries that calculate confidences because Trio crashed.
- We have no measurements for Inlining queries on the largest two documents, because the data could not be imported into Trio.

We consulted one of the developers of Trio, but he also could not resolve these problems for us. We have verified that queries for both techniques return the same results under the same circumstances, so we are confident that we are measuring execution times for queries that do not return bogus results.

Results for the succesful runs can be found in Fig. 10. 'Inl' stands for Inlining; 'XA' for XPath Accelerator; 'Conf' and 'NoConf' for with and without the calculation of confidence scores, respectively. The wobble in the 'Inl NoConf' line of Query 2 is caused by rounding an imprecise measurement (0.03 vs. 0.02).

The first thing that stands out is that it seems that the XPath Accelerator-based approach is much less efficient than the Inlining-based approach. We believe, however, that we cannot draw this conclusion from these results. The XPath Accelerator queries use several intermediate tables which causes much overhead. Furthermore, it also does not permit the query optimizer to globally

optimize the query. Since this query splitting is not inherent to the XPath Accelerator technique, but a measure taken because of practical problems with Trio, it would be unfair to draw this conclusion.

A second observation is that the query execution time does not significantly increase with increasing size except for the largest document for XPath Accelerator. The shape of the curve seems to indicate that evaluation of these queries scales exponentially. Unfortunately, we do not know how much time the Inlining queries would have taken on the largest document. Only for join queries, we see a rise at a size of around 7000 nodes.

A third observation is that is the query execution times do not significantly differ for the three queries. Finally, calculation of confidences is typically an expensive operation. In this application context, however, we see that for XPath Accelerator the overhead for calculating the confidences is relatively negligible.

8 Conclusions

In this paper, we explored the feasibility of storing and querying probabilistic XML using an uncertain relational database. Our approach is to adapt existing techniques for mapping XML data to relational data. We showed how to do this for two representative techniques, namely a schema-based (inlining) and a schemaless one (XPath Accelerator). The key is to make sure that the result represents the same possible worlds as the original probabilistic XML document. In this way, no adaptation in the translation of XML queries is needed. The space complexity is the same as for the underlying mapping techniques.

The maturity of probabilistic relational databases also influences the feasibility. We investigated this by experimenting with a few queries on mapped data on one of the state-of-the-art probabilistic database systems, called Trio. Unfortunately, we encountered some problems with loading the mapped data and with calculating confidence scores for query results. Based on the experiments that did run smoothly or for which we could find a workaround, we observed, for example, exponential scaling for queries on data resulting from mapping XML with the adapted XPath Accelerator technique. On the other hand, confidence calculation proved relatively inexpensive here. Queries on mapped data from the adapted inlining technique appear to be more efficient, but loading mapped data from larger documents and confidence computation failed with this technique.

In this research, we only touched the surface by focussing on feasibility of the approach. For future work we, first of all, intend to expand our experiments to other probabilistic database systems to see if our conclusions hold in general. We also like to compare this approach to extending existing XML databases with support for probabilistic XML. It would be scientifically worthwhile to formally prove that the data and query mapping to the relational domain are indeed correct with respect to XPath semantics and possible world theory. It is also likely that such a formal investigation uncovers opportunities for query optimization. Finally, we intend to turn this work into a benchmark for probabilistic databases.

References

1. Abiteboul, S., Kimelfeld, B., Sagiv, Y., Senellart, P.: On the expressiveness of probabilistic XML models. The VLDB Journal **18**(5) (2009) 1041–1064
2. Huang, J., Antova, L., Koch, C., Olteanu, D.: MayBMS: a probabilistic database management system. In: Proc. of SIGMOD, Providence, Rhode Island, USA, June 29 - July 2. (2009) 1071–1074
3. Antova, L., Koch, C., Olteanu, D.: MayBMS: Managing incomplete information with probabilistic world-set decompositions. In: Proc. of ICDE, Istanbul, Turkey. (2007) 1479–1480
4. Mutsuzaki, M., Theobald, M., de Keijzer, A., Widom, J., Agrawal, P., Benjelloun, O., Sarma, A.D., Murthy, R., Sugihara, T.: Trio-One: Layering uncertainty and lineage on a conventional DBMS (demo). In: On-Line Proc. of CIDR, Asilomar, CA, USA, January 7–10, www.crdrdb.org (2007) 269–274
5. Benjelloun, O., Sarma, A.D., Hayworth, C., Widom, J.: An introduction to ULDBs and the Trio system. IEEE Data Engineering Bulletin **29**(1) (2006) 5–16
6. Boulos, J., Dalvi, N., Mandhani, B., Mathur, S., Re, C., Suciu, D.: MYSTIQ: a system for finding more answers by using probabilities. In: Proc. of SIGMOD, Baltimore, Maryland, USA. (2005) 891–893
7. Cheng, R., Singh, S., Prabhakar, S.: U-DBMS: A database system for managing constantly-evolving data. In: Proc. of VLDB, Trondheim, Norway. (2005) 1271–1274
8. Hung, E., Getoor, L., Subrahmanian, V.: PXML: A probabilistic semistructured data model and algebra. In: Proc. of ICDE, Bangalore, India. (2003) 467
9. Abiteboul, S., Senellart, P.: Querying and updating probabilistic information in XML. In: Proc. of EDBT, Munich, Germany. (2006) 1059–1068 LNCS 3896.
10. van Keulen, M., de Keijzer, A., Alink, W.: A probabilistic XML approach to data integration. In: Proc. of ICDE, Tokyo, Japan. (2005) 459–470
11. Kimelfeld, B., Kosharovsky, Y., Sagiv, Y.: Query evaluation over probabilistic XML. The VLDB Journal **18**(5) (2009) 1117–1140
12. Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D.J., Naughton, J.F.: Relational databases for querying XML documents: Limitations and opportunities. In: Proc. of VLDB, Edinburgh, Scotland, UK. (1999) 302–314
13. Grust, T.: Accelerating XPath location steps. In: Proc. of SIGMOD, Madison, Wisconsin. (2002) 109–120
14. Murthy, R., Banerjee, S.: XML Schemas in Oracle XML DB. In: Proc. of VLDB, Berlin, Germany. (2003) 1009–1018
15. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: Proc. of SIGMOD, Chicago, IL. (2006) 479–490
16. van Keulen, M., de Keijzer, A.: Qualitative effects of knowledge rules and user feedback in probabilistic data integration. The VLDB Journal **18**(5) (2009) 1191–1217
17. Grust, T., Rittinger, J., Teubner, J.: Pathfinder: Xquery off the relational shelf. IEEE Data Engineering Bulletin **31**(4) (2008) 7–14
18. van Keulen, M., de Keijzer, A.: Qualitative effects of knowledge rules in probabilistic data integration. Technical Report TR-CTIT-08-42, CTIT, Univ. of Twente, Enschede, The Netherlands (2008) ISSN 1381-3625.