Master Thesis
University of Twente

Querying Probabilistic XML

Ruud van Kessel

Supervisors:
Dr. ir. Ander de Keijzer
Dr. ir. Maurice van Keulen
Dr. Maarten Fokkinga

Enschede, April 2008

Management Summery

In the scientific field and in working with data integration, uncertain data is a very common subject. In [KKA05] a compact representation is proposed for storing uncertain data in XML. A naive way of querying this data is by calculating all possible worlds and execute the query on each of the worlds. Calculating these possible worlds is however very inefficient because of the exponential growth of worlds. In this thesis we will investigate how the compact representation can be queried in an efficient way.
We will compare two methods for querying the compact representation: Recursive path analysis and the Compare paths method.

Recursive path analysis.
Using a script, for each step of the query a piece of XQuery code is generated, which returns each possible answer for that step. The output of step one is the input of step two and so on. The increase in performance is obtained by calculating the possible answers for the query, instead of calculating all possible worlds for the document.

Compare paths method.
The query is converted by adding the needed possibility and probability steps to the query. When executing queries that include a predicate, an extra check has to be performed to examine if the returned results indeed can occur together with one of the elements referred to in the predicate. We do this by comparing the paths of node identifiers belonging to the probability and possibility ancestors of the candidate elements with the ones of the predicate elements. Two elements occur in the same world only if the number of probability ancestors that occur in both paths of the two elements is equal to the number of probability ancestors that occur in both paths.

We test both methods by executing several queries on test documents of different sizes and containing different levels of uncertainty. This leads to the following conclusions:
- Even for large documents (up to an address book containing 1000 people) the compare paths method works well. However when requesting documents with a lot of descendants in the result, the performance decreases quickly. This is a point of interest for future work.
- The performance of the recursive path analysis is more dependent of uncertainty. Therefore it works better on smaller documents and documents with a smaller level of uncertainty.
- In the recursive path analysis, no feature of checking the correctness of child nodes is implemented. For this reason it performs better than the compare paths method when elements with a lot of children are returned. However, when using predicates there are several cases in which the result can contain incorrect child nodes because simply every node is returned.

Contents

1 Introduction

1.1 Motivation

In our modern world, stored data is everywhere around us. Just think of the client databases of your bank, insurance company or hospital, but also of the geographical data in a navigation system or the contacts in your mobile phone. In most cases this data is stored in a relational database, because of the clear table structure and the fast lookup methods these databases offer.
In several cases, however, it is preferred to represent data as a graph instead of using tables. For example when the structure of the data changes frequently. XML is the most commonly used standard to represent this semistructured data. In 2000 [CFP00] stated that data representation, data interchangeability and the abilities of using XML as a repository are three promising perspectives of XML. Nowadays XML is used more and more instead of HTML for representing web-pages. Furthermore, it is widely used for RSS-feeds and it is the basis in the SOAP protocol for exchanging messages between web-services.
Semistructured data storage systems like [BGK+06], [DAF04], [DFS99], [FK99], [KKR+00], and [JAC+02] are used for storing and querying XML documents. In most cases, this is done by mapping XML to relational tables.
What is common for relational databases and the current semistructured data storage systems is, that the data stored is assumed to be the correct data. If the database of your bank for example contains a customer "John" with account number "1234567", then you can assume there is a "John" with such an account i.e. the data is certain.
There are however several cases in which the data one obtains, is somehow not certain. To illustrate the need for the possibility to store uncertain data, we will give a few examples in the following sections.

1.1.1 Applications

In the scientific field all kinds of experiments are executed. In many cases this leads to uncertain data. For example, sensors produce inherently uncertain data, because sensors usually return a value with a certain inaccuracy, instead of one precise value. Manipulating sensor data probably produces uncertain results as well.
[NJ02] gives an example of uncertainty in scientific data by giving insight into the area of proteomics. A challenge in this area is to identify individual proteins. For this task several experimental tests are available, all with varying reliability. Cases may occur in which proteins are totally misidentified. For following steps in the process an efficient way of storing the level of uncertainty of the test is crucial. Working with imprecise sensors and running test programs that may deliver multiple results, are common sources of uncertain data in the scientific field.

Another example is a speech recognition system that could return several options for

processed spoken words. The system may have recognized that you said “Hello”, but it might also have been “Yellow” (see Figure 1.1). In such a system it is possible that one wants to store both values and return those to the user for feedback and interactive learning of the system.

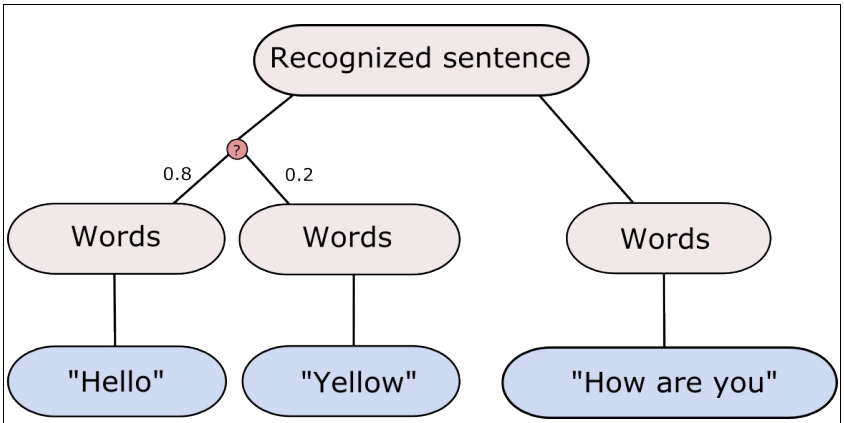


Figure 1.1: Schematic overview of possible speech recognition output

A system that is related to the one described above, but used for more serious business, is the military surveillance system described in [HGS03]. In this case, instead of spoken language, images of a battlefield are processed. These images may contain several objects that need to be classified, for example vehicle convoys or refugee groups. It is not always possible to extract precise information like the exact number of refugees or the specific type of vehicle in a convoy, but all different possibilities need to be stored to create an overview of the situation, so that important decisions can be based on this data.

1.1.2 Data integration

Besides the uncertainty in external information, uncertainty can also occur when integrating two or more certain data sources. This may become clear when combining, for example, the address book stored on your computer with the one stored on somebody else's laptop. There may be contacts that you both know. But if the contact “John” has “john@hotmail.com” as an email address in your address book and the other person knows a “John” with “john@gmail.com”, then which email address is right and are we even talking about the same “John”?

These uncertainties are hard to store in a normal database. Therefore methods are currently being investigated to adapt traditional databases in such a way that it is possible to store uncertain data. This has been done for relational databases but also for XML databases.

1.2 Problem

We have shown that systems that are able to store uncertain data, have an important role to fulfill. Probabilistic databases differ from normal databases in the following way. A normal database describes one world in which all data is certain. Because in a probabilistic database the data contains different possibilities, instead of one world, multiple possible worlds are described. Hence, a probabilistic database can be seen as a collection of several normal databases that each describe a possible world. However, storing probabilistic data this way is very inefficient, because the number of possible worlds grows exponentially with the number of possibilities that the document contains. Therefore a probabilistic database uses a *compact representation* for storing probabilistic data. In this thesis, we investigate how we can query this compact representation in an efficient way.

1.3 Problem definition

The problem is defined as follows:

How can we efficiently query probabilistic XML documents in the compact representation, in such a way that we get the correct result including the associated probabilities?

1.3.1 Goals

In [*KKA05*] the theory behind querying probabilistic data is explained. A naive implementation of this theory implies constructing all possible worlds and executing a query on each of them. This is an inefficient process, because the number of possible worlds grows exponentially with the number of possibilities in the document. Our goal is to improve this situation by developing a technique to process queries on a probabilistic XML document in an efficient manner.
Normal queries on XML documents are formulated in XQuery or XPath. Our goal is to support a significant subset of XPath.

1.3.2 Research questions

To guide this research to a successful solution for our main problem, we formulated the following research questions:

- *Which alternatives are known for querying probabilistic data?*
  We want to know if there currently exist methods that are useful for our research. How is the querying of probabilistic data solved for relation databases and what research is done in the semisturctured field?
- *How does a probabilistic XML document differ from a normal XML document and how does this affect the execution of a query?*
  The structure of a probabilistic XML document differs from a normal XML document. Therefore, it is possible queries have to be converted to another

format. We want to know to what extend this influences the total process of executing a query.

- *On what properties of the compact representation should an approach focus for more efficiently query evaluating on probabilistic documents?*
  Instead of querying all possible worlds independently, we want to query a compact representation. The properties of this representation are different which possibly creates the opportunity to evaluate queries in an new and efficient way.
- *When converting a query to a probabilistic format, should we look at step level or at the query in total?*
  XPath queries exist of different steps. Is it possible to convert them one by one? Is it possible to convert the query as a whole? And where in this process i.e. Between steps or at the and, should the query actually be evaluated.
- *How do alternative approaches for querying probabilistic data compare concerning efficiency?*
  When having different set of (probabilistic data) which approach performs best for which set? Does the format of the query itself influences this result?
- *How should answers of a query on a probabilistic XML document be represented?*
  The representation of a probabilistic query result differs from a normal result, because the results occur with a certain probability. What extra information is necessary to include in the result? What kind of styles can be thought of to represent this extra data?

1.3.3 Research method

By analyzing the properties of probabilistic XML documents in the representation with probability and possibility nodes, we create a prototype mostly written in XQuery. We do performance experiments to test the prototype's efficiency. We compare the prototype with other methods to query probabilistic data, including the naive approach. We use these comparisons to verify to what extent our goal is reached.

1.4 Overview

We continue in Chapter 2 with the related research done in this field. In Chapters 3, 4 and 5 we discuss three different ways of querying probabilistic data: the naive approach, recursive path analysis and the compare paths method respectively. In Chapter 6 we present our experiment evaluation and we take a look at the results of these experiments. In Chapter 7 we will formulate an overall conclusion and recommendations for future work.

2 Background & related research

To understand exactly what this investigation is about, we will explain in this chapter some important concepts in this chapter. First we will show how uncertain data can be considered as a description of multiple possible worlds. After this, we give an overview of the representation method used in [KEI06],[KKA05], because we use this way of representing uncertain data in the rest of this report. We conclude by explaining how specific information is normally extracted from a database and what the difference is when querying probabilistic data.

2.1 Possible worlds

In working with probabilistic data whereby mutual exclusive possibilities can occur, is it useful to keep in mind the possible world semantics: the idea that an uncertain document can be seen as a sequence of possible worlds. When we have somehow retrieved uncertain data, this means that we are uncertain about what elements occur in our world. So instead of listing one certain world, we list all possible worlds, together with (for each of them) the probability it that is the correct one. In general an uncertain document does not describe all of these possible worlds separately, but uses a compact representation using less storage space.
Looking at our speech recognition system (see Figure 2.1) again, it can be seen that the corresponding XML document describes the world in which "Yellow" or "Hello" can be said, followed by "How are you". We can look at this one uncertain world as if it exists of two possible worlds: the world in which is said "Yellow, how are you" and the world in which is said "Hello, how are you". Only one of those possible worlds is the correct one. At this point, we don't know which one, but we estimate that with a probability of 0.8 the "Hello"-world is correct against a 0.2 probability of the "Yellow"-world.

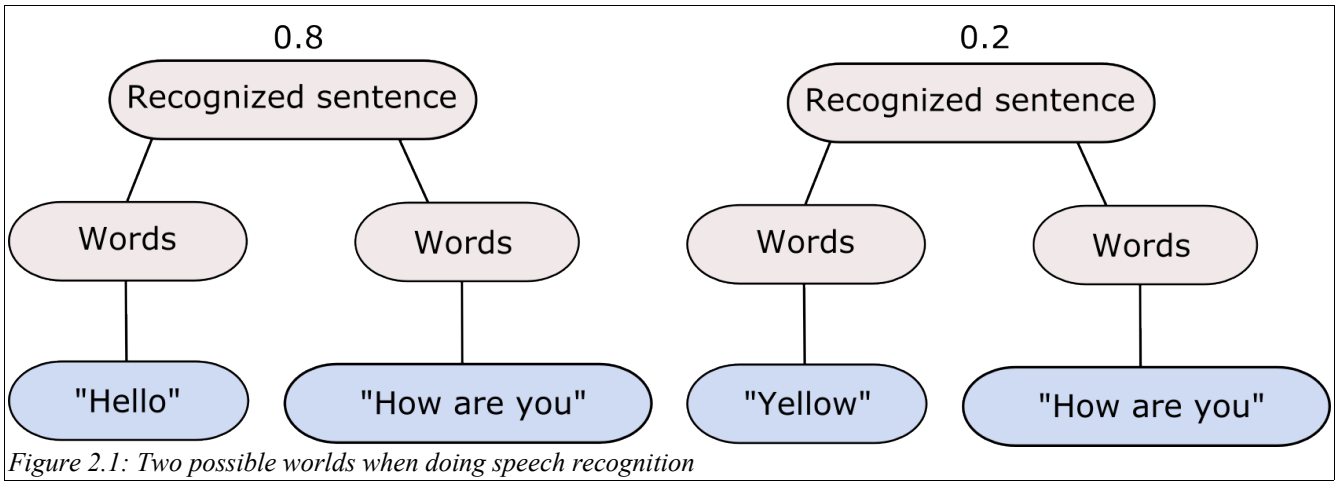


Figure 2.1: Two possible worlds when doing speech recognition

2.2 Representation of uncertain data

We have seen that working with uncertain data leads to different possible worlds. The number of possible worlds grows exponentially with every possibility. If we manage to store every world separately in a database, the size of our database will grow exponentially too. Using a more compact representation is attractive because of the possibility to use less space for storing the data. The several relational and semistructured applications use different representations for storing probabilistic data in a compact way.
For instance, Trio[WID05], a relational probabilistic database project of the Stanford university, uses an uncertainty and lineage database (ULDB) filled with x-tuples. These x-tuples can be seen as normal tuples with the addition that for each element in the tuple more than one alternative can be given. These alternatives are mapped onto a regular relational table.
MayBMS [AKO07] is another relational probabilistic database system comparable with Trio. In this system the compact representation of possible worlds is called a world-set decomposition (WSD). Instead of one table with more (possible) attributes in one tuple, multiple tables with tuples containing one attribute are created for every group of possible attributes. Alternative representations like world-set decomposition templates (WSDTs) and unified world-set decomposition templates (UWSDTs) are used to reduce the number of tables in the database.
To reduce the space needed for data storage the probabilistic XML application ProTDB [NJ02] specifies some special nodes and attributes that are inserted into the XML document to indicate the presence of uncertainty. For every normal element an attribute "prob" (standing for probability) can be added which has a certain value between 0 and 1. Also a "val" (for value) element with a "prob" attribute can be added to indicate that all nodes contained in this "val" element have some probability to occur. One or more of those "val" elements have to be placed inside a "dist" element. This "dist" element has an attribute "type" that describes the distribution type of the underlying "val nodes" and which can be "mutual-exclusive" or "independent".
To store uncertainties in XML documents [KKA05] introduces its own compact representation comparable with the one used by ProTDB. The main difference is that the method described in [KKA05] holds that every distribution is mutual exclusive. This is achieved by introducing two extra elements with a special meaning: the probability node and the possibility node. A probability node is used to indicate that there could be multiple mutually exclusive possibilities present under that node. A possibility node is used to indicate that the underlying node has a certain chance to occur, identified with the value attribute (called "prob") of this possibility element. Every normal node is preceded by a probability node and a possibility node.
A piece of sample probabilistic XML with a possible output of the speech recognition system mentioned in 2.1 is shown in Figure 2.2:

```
<prob>
  <poss prob="1">
    <recognizedsentence>
      <prob>
        <poss prob="0.8">
          <words>Hello</words>
        </poss>
        <poss prob="0.2">
          <words>Yellow</words>
        </poss>
      </prob>
      <prob>
        <poss prob="1">
          <words>How are you</words>
        </poss>
      </prob>
    </recognizedsentence>
  </poss>
</prob>
```

Figure 2.2: Probabilistic XML document for speech recognition

We will continue to use this representation in the next coming parts of this report, because this work builds on the method introduced by [KKA05].

2.3 Querying data

To get specific data from a database a query has to be given as input. In general a query contains information about the location where the data can be found and about the conditions that the data to be returned will have to fulfill. For a relational database SQL is a common query language. For a specific table, elements of rows can be returned when a row fulfills a certain condition.

Students		
studentnr	*name*	*city*
10	Jan	Enschede
11	Henk	Enschede
20	Piet	Amsterdam

To get for example the names of the students that live in Enschede one can execute the following SQL-query.

```
SELECT name FROM students WHERE city="Enschede";
```

This leads to the following result:

Result
name
Jan
Henk

However, XML documents do not work with tables, but instead they have the structure of a tree (see Figure 2.3 with its tree representation in Figure 2.4). Therefore new query languages have been developed, for querying XML documents. The XQuery [CFR+00] standard together with the XPath [CD99] standard (which is a subset of XQuery) are widely used. In the query one describes the path where the needed information is located in the XML tree.

```
<students>
        <student>
                <studentnr>10</studentnr>
                <name>Jan</name>
                <city>Enschede</city>
        </student>
        <student>
                <studentnr>11</studentnr>
                <name>Henk</name>
                <city>Enschede</city>
        </student>
        <student>
                <studentnr>20</studentnr>
                <name>Piet</name>
                <city>Amsterdam</city>
        </student>
</students>
```

Figure 2.3: students.xml

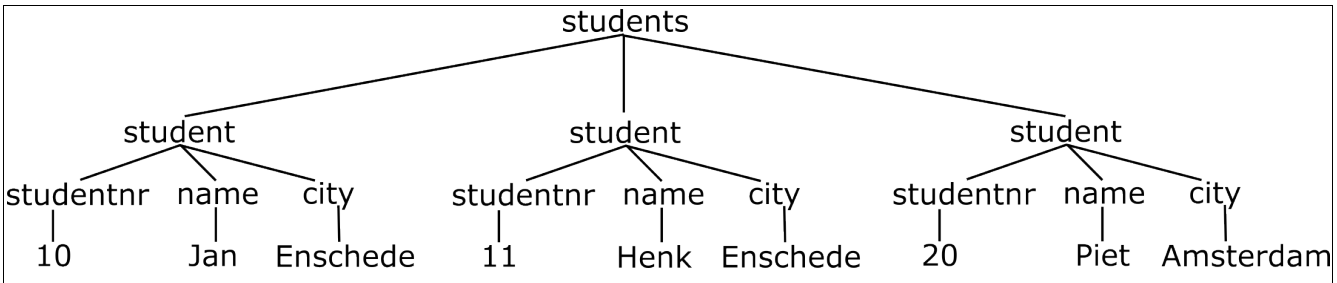


Figure 2.4: students.xml represented as tree.

We give the XPath query for selecting the names of the students that study in Enschede. We give the path where to find the "name" node (Figure 2.5), but because we only want the name of those students who live in Enschede, we add a predicate to "student" in Figure 2.6.

```
doc("students.xml")/students/student/name
```

Evaluation steps:

- get the document student.xml
- from this result, get all the underlying "students" elements
- from this result, get all the underlying "student" elements
- from this result, get all the underlying "name" elements

Result:

```
<name>Jan</name>,
<name>Henk</name>,
<name>Piet</name>
```

Figure 2.5: Evaluation of doc("student.xml")/students/student/name

```
doc("students.xml")/students/student[./city="Enschede"]/name
```

Evaluation steps:

- get the document student.xml
- from this result, get all the underlying "students" elements
- from this result, get all the underlying "student" elements
- from this result, get only those "student" elements that contain an element "city" with the text "Enschede"
- from this result, get all the underlying "name" elements

Result:

```
<name>Jan</name>,
<name>Henk</name>
```

Figure 2.6: Evaluation of doc("student.xml")/students/student[./city="Enschede"]/name

The XQuery language is a lot more complicated than this, but XPath queries of this kind are the ones we pay most attention to in this report.

After the creation of a compact representation of all possible worlds a new problem arises: how do we query this representation? In the Trio system this problem is tackled by introducing TriQL, which is an extension of SQL. These TriQL queries can be executed on the ULDB containing x-tuples. This is done by parsing the TriQL statements which results in one or more SQL queries that are executed on the tables containing the possible alternatives.
The strategy of using some kind of module that converts a query into a suitable format for the probabilistic representation is also used by MayBMS. This system uses relational algebra. New versions of select, product join and other functions are created to query the multiple tables of their WSD.

The goal of the current investigation is to make it possible to execute queries like the

ones described in Figure 2.5 and Figure 2.6 directly on the compact representation of a probabilistic XML document (like the one shown in Figure 2.2). When we query a probabilistic document this can be seen as executing a query on every possible world. For the compact representation the approach of evaluating the query and the final representation style may be different compared to querying all possible worlds separately. However, the final answer should correspond with the answer that would have been returned when each possible world was queried separately. We have seen in section 2.2 that the compact representation contains probability and possibility nodes which we do not want to take into account when formulating our query. This goal corresponds with the one described for the ProTDB system where the "dist" and "val" nodes should not be specified in the query itself, but probabilities should be returned in the result. To accomplish this the query parser module and the query evaluator module of Timber are adapted. The query parser module is changed in such a way that "dist" and "val" nodes are inserted where needed, before executing the query. The query evaluator takes care of the probability calculations for the result.
The syntax problem (handled by the query parser module in ProTDB) is only a sub-problem we have to deal with when querying the compact representation. A bigger issue is the huge number of calculations that have to be done to find each of the possible answers. The representation styles for the final result (described in the next subsection) play an important role when dealing in possible answers. In chapters 4 till 6 different solutions for the total problem are described. The amount of attention paid to the syntax part is different for each solution.

```
<prob>
   <poss prob="1">
     <addressbook>
        <prob>
          <poss prob="0.5">
            <person>
              <prob>
                <poss prob="0.7">
                  <phones>
                    <prob>
                      <poss prob="1">
                        <homephone>1111</homephone>
                      </poss>
                    </prob>
                  </phones>
                </poss>
                <poss prob="0.3">
                  <phones>
                    <prob>
                      <poss prob="1">
                        <homephone>2222</homephone>
                      </poss>
                    </prob>
                    <prob>
                      <poss prob="0.75">
                        <homephone>2323</homephone>
                      </poss>
                      <poss prob="0.25">
                        <homephone>2424</homephone>
                      </poss>
                    </prob>
                  </phones>
                </poss>
              </prob>
            </person>
          </poss>
          <poss prob="0.5">
            <person>
              <prob>
                <poss prob="1">
                  <phones>
                    <prob>
                      <poss prob="0.8">
                        <homephone>3434</homephone>
                      </poss>
                      <poss prob="0.2">
                        <homephone>3535</homephone>
                      </poss>
                    </prob>
                    <prob>
                      <poss prob="0.5">
                        <homephone>3636</homephone>
                      </poss>
                      <poss prob="0.5">
                        <homephone>3737</homephone>
                      </poss>
                    </prob>
                  </phones>
                </poss>
              </prob>
            </person>
          </poss>
        </prob>
     </addressbook>
   </poss>
</prob>
```

Figure 2.7: Probabilistic XML document for an addressbook

2.4 Result representation styles

This investigation is aimed at developing a tool that is able to query the compact representation of a probabilistic XML document in a more efficient way. Before a prototype can be build a representation style has to be chosen for our query output. In the coming subsections we describe three possible representation styles. Each representation style is illustrated by an example. Those examples show the output of the particular representation style when executing the following query on the document shown in Figure 2.7:

```
doc("figure2.7")/addressbook
/person/phones/homephone
```

```
<prob>
  <poss prob="0.35">
    <homephone>1111</homephone>
  </poss>
  <poss prob="0.1125">
    <homephone>2222</homephone>
    <homephone>2323</homephone>
  </poss>
  <poss prob="0.0375">
    <homephone>2222</homephone>
    <homephone>2424</homephone>
  </poss>
  <poss prob="0.2">
    <homephone>3434</homephone>
    <homephone>3636</homephone>
  </poss>
  <poss prob="0.2">
    <homephone>3434</homephone>
    <homephone>3737</homephone>
  </poss>
  <poss prob="0.05">
    <homephone>3535</homephone>
    <homephone>3636</homephone>
  </poss>
  <poss prob="0.05">
    <homephone>3535</homephone>
    <homephone>3737</homephone>
  </poss>
</prob>
```

Figure 2.8: All possible worlds style

2.4.1 Possible worlds style

This representation style (shown in Figure 2.8) is the style that corresponds with the naive method of querying the compact representation of probabilistic data. The result is now represented as answer per possible world. In a possible world all elements can be seen as certain because the world itself has a probability to occur. The probability of an answer equals the probability of the possible world on which the query is executed.

It can be seen that answers are given as combinations of nodes and the correct answer is one of the given combinations. The probabilities of all combinations add up to 1 which is intuitively correct. The number of answers increases however exponentially as the number of possibility-nodes that are involved in the answer grows.

```
<prob>
  <poss prob="0.5">
    <seq>
       <prob>
          <poss prob="0.7">
             <homephone>1111</homephone>
          </poss>
          <poss prob="0.3">
             <subseq>
                <prob>
                   <poss prob="1">
                      <homephone>2222</homephone>
                   </poss>
                </prob>
                <prob>
                   <poss prob="0.75">
                      <homephone>2323</homephone>
                   </poss>
                   <poss prob="0.25">
                      <homephone>2424</homephone>
                   </poss>
                </prob>
             </subseq>
          </poss>
       </prob>
    </seq>
  </poss>
  <poss prob="0.5">
    <seq>
       <prob>
          <poss prob="0.8">
             <homephone>3434</homephone>
          </poss>
          <poss prob="0.2">
             <homephone>3535</homephone>
          </poss>
       </prob>
       <prob>
          <poss prob="0.5">
             <homephone>3636</homephone>
          </poss>
          <poss prob="0.5">
             <homephone>3737</homephone>
          </poss>
       </prob>
    </seq>
  </poss>
</prob>
```

Figure 2.9: Document structure style

2.4.2 Document structure style

This style (see Figure 2.9) can be seen as an improved version of the possible worlds style. The underlying concept is not to display all possible worlds but to keep the result nodes in the same structure as used for the original document. Thus the structure is quite compact. Furthermore, knowledge about the possibilities is preserved, so that it is clear which combinations of result nodes form an answer.
However a "prob" node cannot be placed directly again under a "poss" node according the compact representation syntax of [KKA05], nodes like, for example, "seq" and "subseq" need to be added to preserve a correct structure of the answer.

```
<result>
  <resultnode val="0.35">
     <homephone>1111</homephone>
  </resultnode>
  <resultnode val="0.15">
     <homephone>2222</homephone>
  </resultnode>
  <resultnode val="0.1125">
     <homephone>2323</homephone>
  </resultnode>
  <resultnode val="0.0375">
     <homephone>2424</homephone>
  </resultnode>
  <resultnode val="0.4">
     <homephone>3434</homephone>
  </resultnode>
  <resultnode val="0.1">
     <homephone>3535</homephone>
  </resultnode>
  <resultnode val="0.25">
     <homephone>3636</homephone>
  </resultnode>
  <resultnode val="0.25">
     <homephone>3737</homephone>
  </resultnode>
</result>
```

Figure 2.10: Possibility-per-node style

2.4.3 Possibility per node style

In this method the answer is represented by each result node with its own possibility instead of the combinations of result nodes that constitute an answer (see Figure 2.10).
Therefore knowledge about the probabilities is lost because there is no way of reconstructing which nodes have the ability to occur together.
The size of this representation style grows linearly with the number of nodes in the original document that satisfy the query.

3 Naive method

3.1 Basic idea

As stated before our probabilistic data is stored in a compact representation with probability and possibility nodes. According to [KKA05], a naive way to query this probabilistic data is by calculating all possible worlds. Each of these distinctive possible worlds can then be queried as a normal XML document. All possible answers that are created this way taken together, form the total result of the probabilistic query.

3.2 In practice

The following example clarifies the principle of the naive method in which all possible worlds are constructed.

```
<prob>
    <poss>
        <person>
            <prob>
                <poss>
                    <name>henk</name>
                </poss>
            </prob>
            <prob>
                <poss>
                    <phone>1111</phone>
                    <roomnr>1</roomnr>
                </poss>
                <poss>
                    <phone>2222</phone>
                    <roomnr>2</roomnr>
                </poss>
            </prob>
            <prob>
                <poss>
                    <email>henk@hotmail.com</email>
                </poss>
                <poss>
                    <email>henk@gmail.com</email>
                </poss>
            </prob>
        </person>
    </poss>
</prob>
```

Figure 3.1: Simple example probabilistic XML document

```
<person>
        <name>henk</name>
        <phone>1111</phone>
        <roomnr>1</roomnr>
        <email>henk@hotmail.com</email>
</person>
```

Figure 3.2a: First of four possible worlds

```
<person>
        <name>henk</name>
        <phone>2222</phone>
        <roomnr>2</roomnr>
        <email>henk@hotmail.com</email>
</person>
```

Figure 3.2b: Second of four possible worlds

```
<person>
        <name>henk</name>
        <phone>1111</phone>
        <roomnr>1</roomnr>
        <email>henk@gmail.com</email>
</person>
```

Figure 3.2c: Third of four possible worlds

```
<person>
        <name>henk</name>
        <phone>2222</phone>
        <roomnr>2</roomnr>
        <email>henk@gmail.com</email>
</person>
```

Figure 3.2d: Fourth of four possible worlds

Figure 3.1 describes a probabilistic document of a person and his or her characteristics.

We want to execute the following XQuery on this document:
`doc("Figure3.1")/person[phone="1111"]//roomnr`

which is the query to return all the room numbers from those persons that have 1111 as a phone number. On first sight, without paying enough attention to the possibilities, one could think that the result would exist of both room 1 and room 2, since those are the room numbers that can be found under the person element in which "phone" is 1111. Now, let us look at Figure 3.2a till 3.2d that describe the 4 possible worlds that can be constructed out of this document (the prob and poss nodes are omitted for readability). Here we can see that only Figure 3.2a and 3.2c return a result, since the persons in 3.2b and 3.2d have the wrong phone number. Both these correct results however, have room 1 as room number. The correct result of this query on the above probabilistic document therefore is room one.

3.3 Observations

As mentioned before this naive method is not very efficient, especially because of the exponentially growing number of possible worlds. This number increases for each probability node in the original document with a factor equal to the number of possibility nodes in that probability node. An example calculation in [KKL06] gives an indication of the fast growth of the number of possible worlds.
As seen in the example in section 3.2, multiple possible worlds may return the same result. In the example room 1 is returned twice although it can be seen as one answer. Since the possible worlds style is used for representing our result, instead of per element we get the probability per possible combination of elements. Because of all the combinations that have to be listed the size of the result grows very fast.
In practice, the function that creates all possible worlds of a document, returns a "worldlist" element which contains world elements that each represent a possible world. This can be considered a simple variation on the possible world representation style.

4 Recursive path analysis (RPA)

In this chapter we discuss a prototype implementation of a proposed solution [KEU08] for the probabilistic query problem. We describe the basic idea behind this solution and show how the prototype is implemented. In the "observations"-subsection we mention the strong points and the limitations of this method.

4.1 Basic idea

A normal XPath query exists of several steps. Each of these steps has an input and an output. The output of the first step serves as the input of the second one and so on. The total sequence of steps results in the queried information. One of the ideas of the "Recursive path analysis"-method is to evaluate probabilistic queries in the same way. A difference with normal queries is that a suitable intermediate result format has to be chosen. Whilst the input and output of a normal XPath step are both sequences of XML nodes or atomic data, the probabilistic variant should contain somehow information about the probabilities. This information should be in such a format that in the next step again calculations can be performed on these probabilities again. When working out this approach it should be kept in mind that answers always have to fit in a possible world which has some probability to occur. The fact that multiple answers may occur in multiple possible worlds led to an intermediate representation that contains world elements each with its probability as attribute.
Every node has a unique identifier. We use these identifiers as references to store result nodes in the world elements. Figure 4.1 shows us a XML document "addressbook.xml" with node identifiers.

```
(1)  <prob>
(2)     <poss prob="1">
(3)        <person>
(4)           <prob>
(5)              <poss prob="1">
(6)                 <name>henk</name>
                 </poss>
              </prob>
(7)           <prob>
(8)              <poss prob=".6">
(9)                 <phone>1111</phone>
(10)                <roomnr>1</roomnr>
                 </poss>
(12)             <poss prob=".4">
(13)                <phone>2222</phone>
(14)                <roomnr>2</roomnr>
                 </poss>
              </prob>
(15)          <prob>
(16)             <poss prob=".8">
(17)                <email>henk@hotmail.com</email>
                 </poss>
(18)             <poss prob=".2">
(19)                <email>henk@gmail.com</email>
                 </poss>
              </prob>
           </person>
        </poss>
     </prob>
```

Figure 4.1: Simple example probabilistic XML document "addressbook.xml" with node identifiers (nids)

```
<world prob="1">
       <nid>3</nid>
</world>
```

Figure 4.2a: result of doc("addressbook.xml")/person

```
<world prob=".6">
       <nid>9</nid>
</world>,
<world prob=".4">
       <nid>13</nid>
</world>
```

Figure 4.2b: result of doc("addressbook.xml")/person/phone

The pieces of sample XML in Figure 4.2a and 4.2b, are the intermediate results between two steps. Figure 4.2a shows the outcome of a step "doc("addressbook.xml")/person". This step results in one possible world with the one person element (with node identifier "3") in it. This result functions as the input for the next step (in this case "/phone"). To get the phone numbers of the selected persons, for each world each person is evaluated to select his or her phone numbers . Figure 4.2b shows the result when both the person and the phone step are executed. For the selected person two possible phones numbers are found, so the selected two possible worlds are returned as result. Note that only possible worlds are created for phone numbers, in contrast to the naive method in which all possible worlds of the total document are created. The output of our /phone step could now serve as input for a next step or in case this was the last step of the query, the result could be converted to the actual query result. This conversion is done by replacing the "nid" nodes by the elements they actually refer to and by placing an "answer"-node around the result.

To create the possible worlds that correspond with the nodes found for one step a function called *allCombinations* is used. In the above example only two possible phone numbers are found which results in two possible worlds, but dependent on the number of possible answers found, more possible worlds can be created. Because some answers cannot occur in the same world because of their mutual exclusive properties, the allCombinations function only creates worlds with those combinations of nodes that have the ability to occur together.

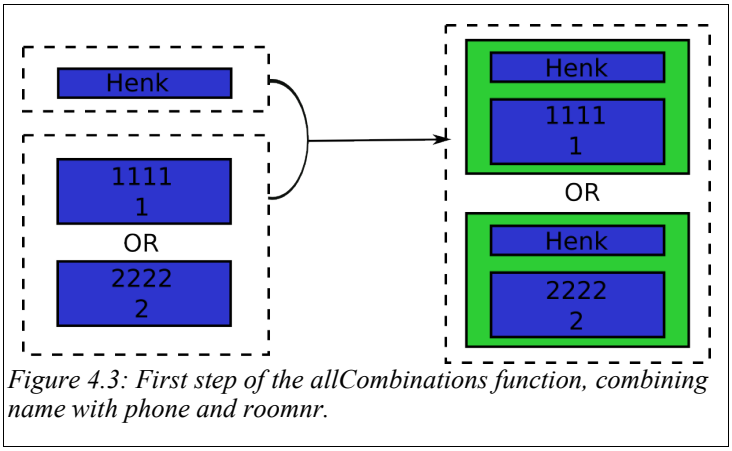


Figure 4.3: First step of the allCombinations function, combining name with phone and roomnr.

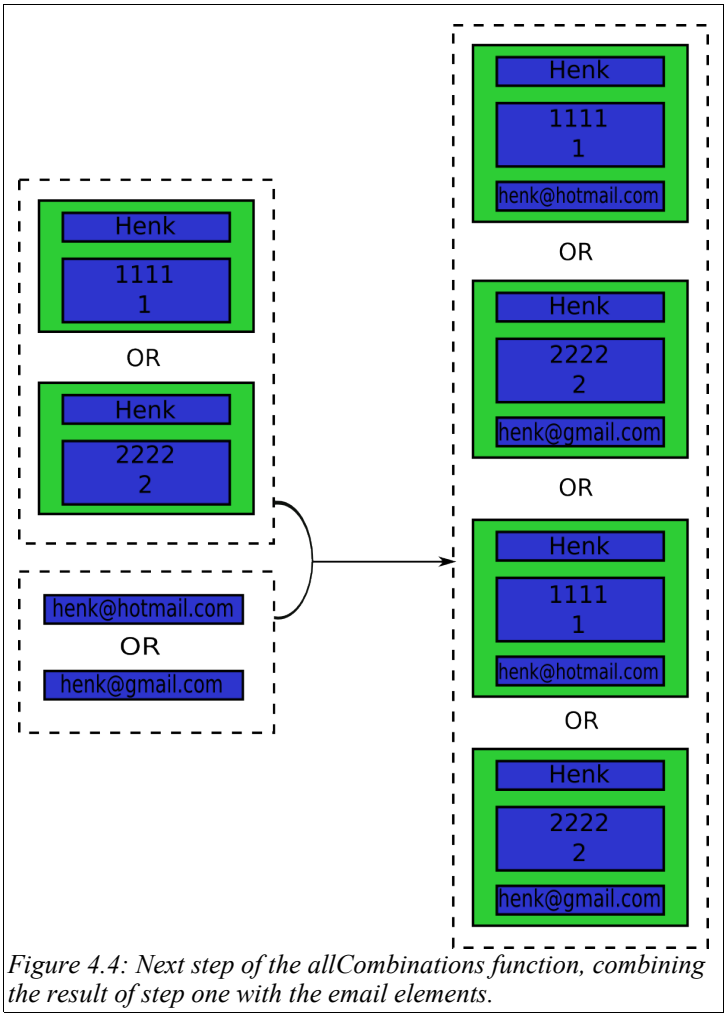


Figure 4.4: Next step of the allCombinations function, combining the result of step one with the email elements.

When we execute the following query (get all child elements from person) on the document of Figure 4.1:

```
doc("addressbook.xml")/person/*
```

The following elements are part of the result:

```
<name>henk</name>,

<phone>1111</phone> <roomnr>1</roomnr> or
<phone>2222</phone> <roomnr>2</roomnr>,

<email>henk@hotmail.com</email> or
<email>henk@gmail.com</email>
```

To get the result according the possible worlds representation style the allCombinations function combines the name element with the possible phone and roomnr elements (see Figure 4.3). This result is then combined with the email elements again (see Figure 4.4), which leads to four possible answers. This method is not only used to represent the final result, but also for the intermediate results. Thus a total possible answer can easily be excluded from the final result if it is found out that in this world a necessary child node doesn't exist.

4.2 In practice

Important in this prototype is, that the prototype itself only generates XQuery (Figure 4.5). The XQuery that is generated works according to the idea explained in section 4.1 and can be executed on the probabilistic XML documents (in our case stored in the MonetDB/XQuery database).

We will first explain how the generated XQuery works and later on we will show how the Perl script manages to generate this XQuery code.

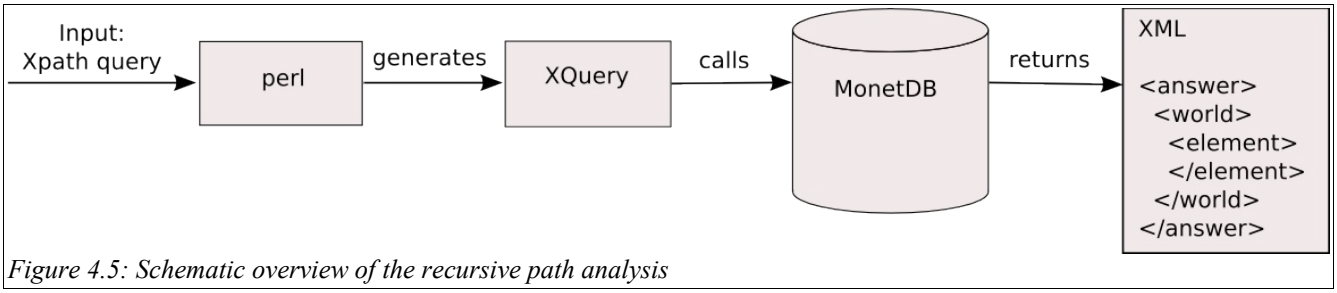


Figure 4.5: Schematic overview of the recursive path analysis

4.2.1 XQuery

The generated XQuery code consists of one function declaration and one big XQuery-statement that uses this supportive function. Every step in the probabilistic XPath query is represented by a piece of XQuery code that has a sequence of world elements

containing node identifiers (nids) as input and as output. The result of one step is used to calculate the result of the following step. The working of the piece of XQuery code that is generated for each step is illustrated in the scheme of Figure 4.7. This scheme shows the step of selecting all phones of (the first) person using the example document listed in Figure 4.6.

```
(1) <prob>
(2)     <poss prob="1.0">
(3)         <person>
(4)             <prob>
(5)                 <poss prob="0.6">
(6)                     <phone>1212</phone>
                    </poss>
(7)                 <poss prob="0.4">
(8)                     <phone>1111</phone>
(9)                     <phone>2222</phone>
                    </poss>
                </prob>
(10)                <prob>
(11)                    <poss prob="1.0">
(12)                        <phone>3333</phone>
                    </poss>
                </prob>
            </person>
(13)            <person>
(14)                <prob>
(15)                    <poss prob="1.0">
(16)                        <phone>4444</phone>
                    </poss>
                </prob>
            </person>
(17)            <person>
(18)                <prob>
(19)                    <poss prob="1.0">
(20)                        <phone>5555</phone>
                    </poss>
                </prob>
            </person>
        </poss>
    </prob>
```

Figure 4.6: Example XML document with nids.

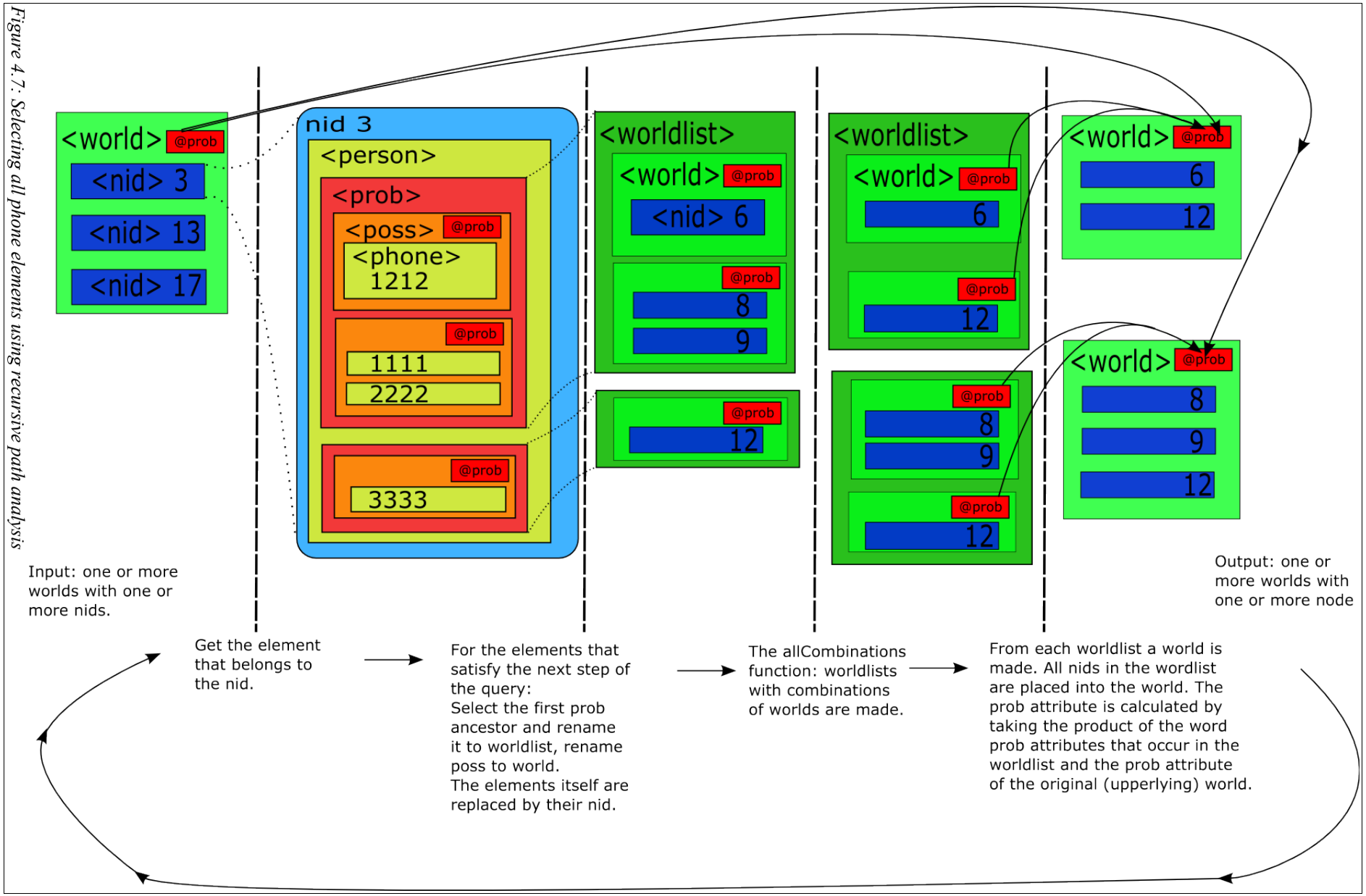


Figure 4.7: Selecting all phone elements using recursive path analysis

In this simplified schema of the process, just one world is shown from which only the element of the first nid is used for evaluation. In practice multiple worlds with multiple nids can be used as input. In the version of MonetDB/XQuery we use, two useful functions are included for handling node identifiers. We use the function pf:nid($element) to get the unique node identifier of an element and we use the function id($nid, $doc) to get the element that belongs to a node identifier (in a certain document).

- We start the process by looking up the elements that belong to the nids in the worlds of our input.
- Now we do the actual selection, in our example we select all "phone" elements together with their probability and possibility parents.
- While selecting them, we convert each probability node to a worldlist node, convert each possibility node to a world node and replace the phone element with its node identifier.
- Now we have zero or more created worldlists for each nid in every world of our original input. The worlds in these seperate worldlists are mutually exclusive, but the worlds inside one worldlist may occur together. The "allCombinations" function creates all worldlists with worlds that may occur together, from the worldlists with mutually exclusive properties.
- We merge all worlds of a worldlist to one world by placing all nids in it.
- We calculate the probability of the new world by taking the product of the probabilities of the merged worlds, and multiplying this value with the probability of the original world from the input. This can be explained as follows: In this last step we create worlds containing elements that occur together. The chances that these elements occur are independent of each other so we take the product of the probabilities of the merged worlds following the rule P(A and B) = P(A)*P(B). The elements to return can only occur when the world they are in actually occurs in the first step. Because the world in the first step itself also has a certain probability to occur we multiply our outcomes with this probability.
- The result: zero or more possible worlds including their probability and having the result-nodes of this query-step as nid elements.

```
1. let $ctx1 :=
2.     for $w1 in $ctx0
3.     let $pw1 := data($w1/@prob)
4.     let $sub1 :=
5.         for $nid1 in $w1/nid
6.         let $xml1 := id($nid1,$scope)
7.         return
8.             for $prob1 in $xml1/prob[./poss/person]
9.             return
10.                <worldlist>{
11.                    for $poss1 in $prob1/poss
12.                    let $newnids1 :=
13.                        for $n in $poss1/person
14.                        return
15.                            <nid>{pf:nid($n)}</nid>
16.                    return
17.                        <world>{$poss1/@prob,$newnids1}</world>
18.                }</worldlist>
19.            )
20.    for $comb1 in allCombinations($sub1)
21.    let $p1 := pf:product($comb1/world/@prob)
22.    return <world prob="{$pw1*$p1}">{$comb1/world/nid}</world>
```

Figure 4.8: Piece of generated XQuery that takes care of a /person step

4.2.2 Perl

As mentioned before the actual prototype is a Perl script that has generating Xquery as its only function. By evaluating the path of the XPath-query in the Perl script, all steps that need to be executed in XQuery can be generated in advance. Although it is unknown what the result of the several steps will be at this point, it is already known which transformations will have to be done on the result. So the Perl-script generates in XQuery, the transformations that have to be done for each step to get the final result. The path of the XPath-query can be evaluated by calling a Perl function for each step. The basic step is the child-step and for each of those the function genstep(input, output, nodetest) is called. The input and output parameter that have to be given contain a number to identify the input and output fur further use. Thus it can be specified that the output of one step is the input of the following step.

```
genstep(0,1,"person");
genstep(1,2,"phones");
genstep(2,3,"homephone");
```

When calling each of these functions, every time a piece of XQuery-code, as listed in Figure 4.8, is generated. In this figure is shown that the first line the output is assigned to the variable ctx1. The second line shows that the result of this person step is obtained by iterating over all items in ctx0: the input. When doing following steps (like phones and homephone) the names of the variables (in the form ctx..*X..*) are changed according to the input and output parameters that are given to the "genstep" function, while the XQuery-code itself stays the same. The Perl-script ends by calling the function genanswer(input). As input the identifier of the last output can be given. This leads to a piece of XQuery-code that converts all nids back to their elements and places the worlds in which those result-elements are located into an answer element.

4.2.3 Predicates

We have seen in the previous section how the prototype handles normal child steps. The prototype is, however, also capable of handling predicates in an XPath-query. Predicates are used to filter out those nodes from the result, that do not fulfill a certain condition. The principle of performing a predicate step is the same as for performing a child step, but the way we handle the input and output is different . When we perform a child step, we would take all the possible worlds generated in the previous step and get the elements belonging to the nids in those worlds. We would select all children of those elements that match our child step and create possible worlds for them.
When we do a predicate step we evaluate each possible world one by one. We get the elements belonging to the nids of the first possible world and check for matches with our predicate for the children of those elements. If those child elements indeed exist we use them to calculate allCombinations. We return the original world with its probability multiplied with the probabilities of the created worlds by the allCombinations function. It is possible that all children of the nids in the original world match the predicate. In that case the original world is returned with a probability of 1. In case there are no children of

the nids in the original world that match the predicate, an empty world with a probability one is returned. In this way every possible world of the previous step is checked for the predicate. The main difference with a normal child step is that the possible worlds of the previous step are evaluated individually in a predicate step. Besides that the output of the allCombinations function of a predicate step is not directly used as result, but is used to modify the possible worlds of the previous step.

4.3 Observations

- Because nids are used in this process, in the end the original elements that correspond with the nids are returned. When an element to return contains children, these children may be in conflict with a predicate given in the query. A practical example of this problem is given in section 5.2.5.
- For every step of the query all possible answers are generated, even if the final step contains a predicate that matches only a few nodes.
- The size of the result is dependent of the number of possible worlds, because of the use of the possible worlds representation style.
- The naive method is improved by calculating only those possible worlds that play a role in the path of the query
- The recursive path of the process (i.e. evaluating what needs to be done of each step of the XPath query), is already done in the Perl-script. This means that the XQuery module doesn't need expensive functions that can handle different lengths of paths.

5 Compare paths method (CPM)

5.1 Basic Idea

The basic idea behind this compare paths method is that we can query probabilistic XML just by replacing all /node steps by /prob/poss/node steps. If we then link the queried nodes with their probability, then we are done. One problem is, we indeed get all the nodes we asked for, but some nodes are not valid. See the following example for an explanation.

When we want to query all roomnr elements in figure 3.1, in a normal XML document we give the following query:

```
doc("something")/person/roomnr
```

Because we work with probabilistic XML we convert this query into:

```
doc("something")/prob/poss/person/prob/poss/roomnr
```

this gives as result:

```
<roomnr>1</roomnr>,
<roomnr>2</roomnr>
```

which is the correct result since we asked for all roomnr elements.

The problem arises when we start using predicates. When we want to have the roomnr of the person that has 1111 as phone number, we would query that in a normal XML document in the following way

```
doc("something")/person[./phone="1111"]/roomnr
```

converting leads to:

```
doc("something")/prob/poss/person[./prob/poss/phone="1111"]/prob/poss/roomnr
```

which gives also as result:

```
<roomnr>1</roomnr>,
<roomnr>2</roomnr>
```

This time the result is incorrect because it contains too much information, because there is no world (see Figures 3.2) in which the phone number "1111" occurs together with room number "2". So our result is not correct until we manage to filter out those nodes that cannot occur in the possible world of the predicate.

Thus, want to determine whether each of the original results (the candidate elements: room numbers "1" and "2") can occur in the same possible world as the predicate. Therefore we do the following query

```
doc("something")/prob/poss/person/prob/poss/phone[.="1111"]
```

which gives us the predicate element: phone number "1111":

```
<phone>1111</phone>
```

Finally a comparison has to be made between the results two both queries.

The general way to get the correct result for an XPath query on the compact representation containing a predicate is by doing the following steps:

1. Get the candidate elements by executing the XPath query (after replacing every /step by /prob/poss/step).
2. Get the predicate elements by executing an XPath query, including prob and poss steps, that returns the elements that are checked for in the predicate of the original query.
3. For every candidate element check whether there exists one or more predicate elements that occur in the same world as the candidate element.
    1. If there is no predicate that occurs in the same world as the candidate element, this candidate element is no part of the result.
    2. otherwise, it is.

Step 1 and 2 were illustrated above. In step 3 of this approach we want to determine whether or not two nodes can occur in the same possible world. We can say the following about this issue:

[1] To check if node1 and node2 occur in the same world, we take all probability and possibility ancestors of both node1 and node2. If, for every probability ancestor of node 1 that is also a probability ancestor of node2, the underlying possibility node is the same for node1 as for node2, then node1 and node2 occur in the same possible world.

What we can conclude from this formulation is the following:

[2] The only case in which node1 and node2 cannot occur in the same possible world is when a probability node exists that is both an ancestor of node1 and node2 but that probability node has different underlying possibility nodes for node1 and node2.

We can simplify this rule to:

[3] if (the number of probability ancestors that is equal for node1 and node2) > (the number of possibility ancestors that is equal for node1 and node2) then: node1 and node2 do not occur in the same possible world (else: they do).

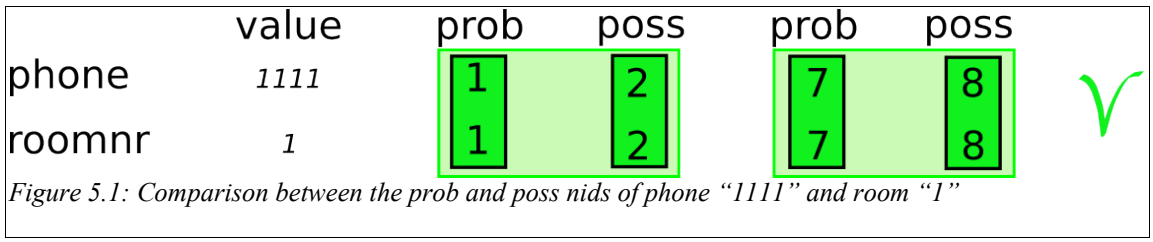


Figure 5.1: Comparison between the prob and poss nids of phone "1111" and room "1"

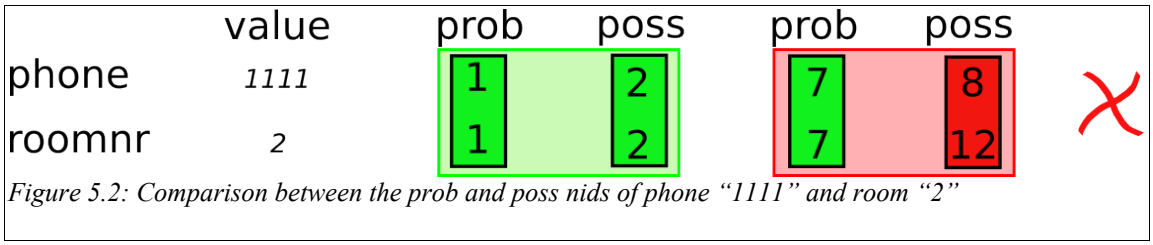


Figure 5.2: Comparison between the prob and poss nids of phone "1111" and room "2"

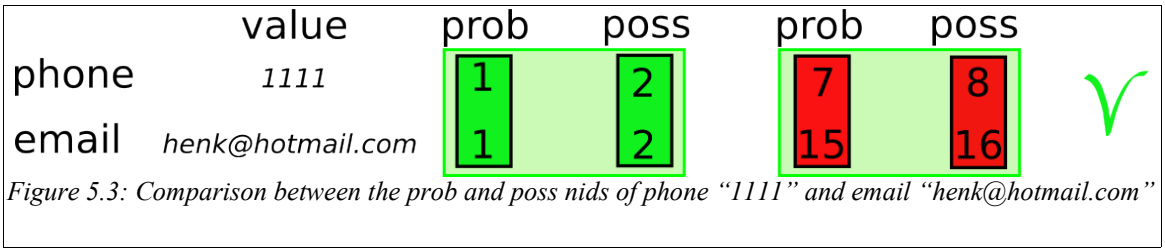


Figure 5.3: Comparison between the prob and poss nids of phone "1111" and email "henk@hotmail.com"

The way in which we use these rules to check whether nodes can occur in the same world is illustrated in Figures 5.1 till 5.3. It can be seen in Figure 5.1 that the roomnr element with value "1" occurs in the same world as the phone element with value "1111" because all their probability and possibility nids correspond.
For the roomnr element with value "2" it is shown in Figure 5.2 that the probability nid corresponds with the probability nid of the phone element while the possibility nid of those both elements differ. This combination leads to the conclusion that the roomnr element with value "2" cannot occur in the same world as the phone number with value "1111".
When the node we want to compare with is located in a totally different probability node, such as the email element with value "henk@hotmail.com" in Figure 5.3, this means that both elements can occur in the same world, as long as the parent elements (person in this case) occur in the same world.

5.2 In Practice

In the following sections it is described how we use the abovementioned ideas as a basis for our prototype.

5.2.1 The representation style of the prototype

First of all, we have to decide what representation style to use for our prototype. We do not select the possible worlds style because of the large number of possible answer combinations that are generated in this style when the level of uncertainty increases. Furthermore, the document structure style is hard to realize because of the "seq" and "subseq" elements that need to be added. Finally, this style does not give a clear overview of the answers because the result nodes may have dependencies with upper laying "seq" or "subseq" nodes.
We choose to use the possibility per node style for our prototype, because of its clear structure and the size of the result that grows linear with the number of queried elements. Another reason is that for simple queries the probability can easily be calculated by multiplying all probability values of the ancestors of the result node which each other.

5.2.2 General overview

In the previous section we introduced three steps to get the right answers in the result. In our prototype we implemented those three steps. The first two (getting the candidate elements and getting the predicate elements) basically mean doing a transformation of the original query. We use a Java parser to perform those transformations.
After we have obtained the candidate and predicate elements, we want to compare paths of nids (the "pps" element in the following examples) to check if candidate elements are part of the final result (as shown in Figures 5.1 till Figure 5.3). For this part of the process we use the functions in our XQuery module.
First of all we call the getnids function for both candidate and predicate elements to get the paths of nids. So, the input of the getnids function is a sequence of (candidate or predicate) elements and the output consists of one nids element containing a completenode element for each of the original input elements (see Figures 5.11 a and b for examples). In this completenode element the original node is listed together with a sequence of the node identifiers of the prob and poss ancestors of this node. These sequences we use for the comparison further on in the process.
The "computeprobs" function that takes care of returning the correct nodes with the correct probability, needs a sequence of nids elements (as generated by the getnids function) as input. The first nids element in the input sequence contains the completenode elements of the candidate elements. Every following nids element in the sequence contains the completenode elements for all the predicate elements that satisfy the predicate. This makes that the length of the nids element sequence used as argument for the computeprobs function is equal to one (for the candidate elements)

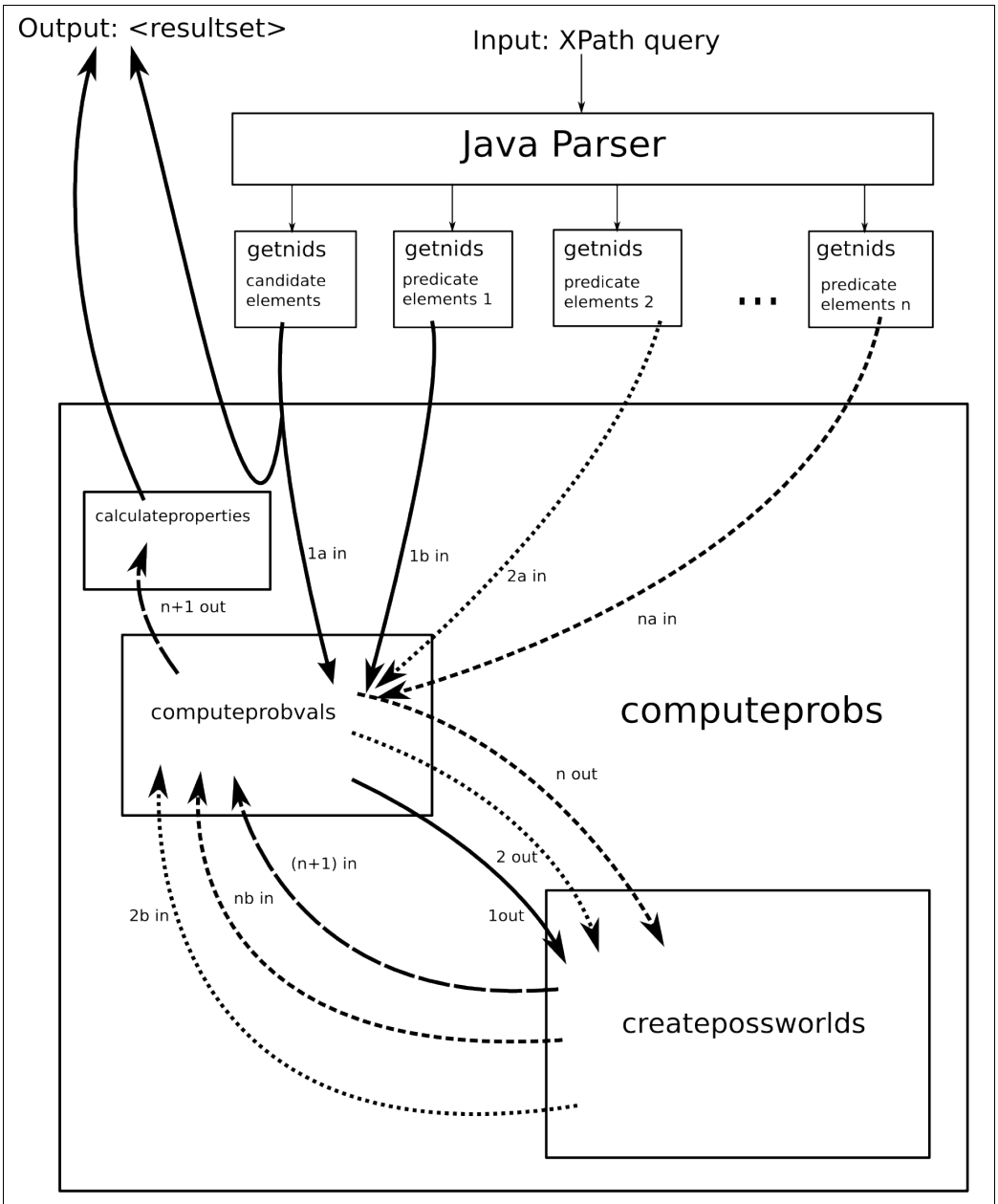
plus the number of predicates used in the query. The computeprobs function returns the results in the possibility-per-node representation style.

In addition to filtering out candidate elements that do not occur in one of the possible worlds of the predicate elements, we also want to give the probabilities that correspond with the elements in the result. The computeprobvals function takes care of the comparisons and calls the createpossworlds function to keep track of the possible combinations of candidate and predicate elements.
To be able to return the right result the computeprobs function should make comparisons of paths of nids and calculate the probabilities of the correct candidate elements. This is done by the following steps (Figure 5.4 illustrates this process including the loop that occurs when following these steps):

1. the computeprobs function calls the computeprobvals function with as argument the nids element that corresponds with the candidate elements, together with the nids element that corresponds with the predicate elements of the first predicate.
2. The computeprobvals function filters out incorrect candidate elements using the before mentioned rules. The probability of an element in the final result is dependent of the probability of the element itself and of the probability of the predicate. (For example, when we query the document in Figure 4.1 for all email addresses, we get the "henk@hotmail.com" with probability 0.8 and "henk@gmail.com" with probability 0.2. However, when we query for all email addresses of those persons that have a phone number "1111", we see that the chance that such person exists is 0.6. So now the overall probability for "henk@hotmail.com" becomes 0.6*0.8=0.48 and for "henk@gmail.com" is will be 0.6*0.2=0.12).
3. To calculate the properties in the right way, we want to know which candidate elements occur in combination with which predicate elements. Therefore the computeprobvals function calls the createpossworlds function, which creates such combinations. Each time a path of nids of a correct candidate element is given as argument, together with the paths of nids of the predicate elements that can occur in the same world.
4. The createpossworlds function takes together these paths of nids in a "pw" element. When there are multiple predicate elements possible that exclude each other multiple "pw" elements are created.
5. It is possible that the original query does contain more than one predicate. If this is the case we return to step 1, but this time we call the computeprobvals function with the "pw" elements found in step 4 instead of the nids element of the candidate elements. For these total "pw" objects, it is now checked if there are predicate elements of the following predicate that occur in their world. We do this to make sure we have no inconsistencies between predicates on different levels. We continue this loop until all predicates of the original query are compared.
6. The computeprobvals function returns for each candidate element one or more "pw" elements. As said before these "pw" elements contain paths of nids for the candidate element together with the paths of nids for one or more predicate elements.
7. For every "pw" element of a candidate element the calculateproperties function is called. This calculates the correct probability of the combination of candidate and predicate elements in the "pw" element.

8. The sum of the probabilities calculated for the “pw” elements of one candidate element is the final probability for that element.

In the following subsections the total process is explained using one simple and one more complicated example.



- "1a in" and "1b in" are the first input of computeprobvals function,
- "1 out" contains the legal combinations of the candidate and predicate elements.
- "2b in" contains the possible world elements which are created for "1out".
- "2b in" together with the second predicate elements "2a in" from the input for computprobvals again.
- This goes on until all predicate elements are compared in computeprobvals.
- Then a last call to createpossworlds is made, the result is used to calculate the final probabilities.
- The final probabilities are returned with the original elements.

Figure 5.4: Total overview of the Compare Paths Method

5.2.3 The Java parser

To prepare XPath queries to be executed by our XQuery module, we built a parser in Java. This parser takes care of the conversion of queries as shown in the first section of this chapter. The module contains a function "computeprobs" which is the core function of the module. This function returns the right nodes with their probability and needs the nodes that are about to get selected together with their probability and possibility nids as input. The "getnids" function creates a list of these nids for each element in the sequence given as argument. The output of the Java parser is shown in Figure 5.5 for the query:

```
doc("addressbook.xml")/person/phone
```

```
1.  import module namespace cpm = "http://monetdb.cwi.nl/XQuery/Documentation/Language/Modules/" at
    "/path/to/cpm.xq";

2.  <resultset>{
3.     cpm:computeprobs(
4.        (cpm:getnids(doc("addressbook.xml")/prob/poss/child::person/prob/poss/child::phone))
5.     )/result
6.  }</resultset>
```

Figure 5.5: XQuery output of the Java parser.

The first line takes care of the import of the XQuery module called "cpm". In line 3, the "computeprobs" function is called. The argument for this function (line 4) is the outcome of the "getnids" function. The "getnids" function gets the converted version of the original XQuery as argument.

When an XPath query with a predicate needs to evaluated, the parser output looks somewhat more difficult. In Figure 5.6 we show the output for the following query:

```
doc("addressbook.xml")/person[./phone="1111"]/roomnr
```

```
1.  import module namespace cpm = "http://monetdb.cwi.nl/XQuery/Documentation/Language/Modules/" at
    "/path/to/cpm.xq";

2.  <resultset>{
3.     for $i0 in
4.     doc("addressbook.xml")/prob/poss/child::person[./prob/poss/child::phone="1111"]
5.     return
6.        cpm:computeprobs(
7.           (cpm:getnids($i0/prob/poss/child::roomnr),
8.           cpm:getnids($i0/prob/poss/child::phone[.="1111"]))
9.        )/result
10. }</resultset>
```

Figure 5.6: XQuery output of the Java parser.

When we look at the lines 6-8, we see that now as argument a sequence of two "getnids" function outcomes is given. The first one is for the candidate elements and returns the probability and possibility nids of all roomnr elements that are located under a person element that also contains a phone element with value "1111". These nids need to be

compared with the nids of the predicate, so the second of the "getnids" calls is executed for the phone elements with the value "1111". In lines 3-6 the "computeprobs" function is called once for every person element that has the specific phone element. This is done because we don't want to mix up child-nodes from different persons in our result, in case that more than one person element with the specific phone number exists.

5.2.4 The CPM XQuery Module

The "computeprobs" function compares the different paths of nids and computes the probabilities of the result. In order to do so, it needs paths of nids generated by the "getnids" function as input. What the "getnids" constructs, is a new element which contains the original node and a sequence of prob-poss-nids pairs with their probability in the following form:

```
<nids>
    <completenode>
        <node>
            <roomnr>1</roomnr>
        </node>
        <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>7</prob><poss>8</poss><value>0.6<value></pp>
            <oavalue>0.6</oavalue>
        </pps>
        <subnodes/>
    </completenode>
    <completenode>
        <node>
            <roomnr>2</roomnr>
        </node>
        <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>7</prob><poss>12</poss><value>0.4<value></pp>
            <oavalue>0.4</oavalue>
        </pps>
        <subnodes/>
    </completenode>
</nids>
```

Figure 5.7: The output of the "getnids" function for two roomnr elements.

In Figure 5.7 the output is given for the "getnids" function called with two roomnr elements (see line 7 of Figure 5.6). For every roomnr element a "completenode" element is constructed. The sub element "node" contains the actual roomnr element. The sub element "pps" (prob-poss-sequence) contains a sequence of probability and possibility nids. These node identifiers are obtained by calling the "pf:nid()" function as explained in section 4.2.1. Every possibility node together with its probability node and the probability value is stored in a "pp" element. The overall value of the node, obtained by multiplying the probability values of every possibility in the path, is stored in the "pps" element.

Now that we executed the "getnids" function for both the candidate elements and the predicate elements, we start comparing each candidate element with each of the

predicate elements in the "computeprobs" function. For each of the candidate elements the "computeprobs" function calls a help function "computeprobvals" that compares the candidate element with every predicate element. If, for every probability nid of the candidate element that is equal to a probability nid of the predicate element, also a possibility nid of the candidate element is equal to a possibility nid of the predicate element then the both nodes occur in the same world and the node can be legally returned (see the rules in section 5.1). Otherwise the two nodes cannot occur in the same world and nothing is returned.
Before returning the legal elements the combination of "pps" elements of the candidate and predicate elements are stored in a "pw" element. This is done in the "createpossworlds" function and requires the ability to calculate the probability of the elements that are part of the final answer. A possible result of the "createpossworlds" function may look as follows (see Figure 5.8):

```
<pw>
    <pps>
        <pp><prob>1</prob><poss>2</poss><value>1</value></pp>
        <pp><prob>7</prob><poss>8</poss><value>0.6</value></pp>
        <oavalue>0.6</oavalue>
    </pps>
    <pps>
        <pp><prob>1</prob><poss>2</poss><value>1</value></pp>
        <pp><prob>16</prob><poss>17</poss><value>0.8</value></pp>
        <oavalue>0.8</oavalue>
    </pps>
</pw>
```

Figure 5.8: The output of the "createpossworlds" function

In the example of Figure 5.8 the outcome of the "createpossworlds" function is given when the phone element with value "1111" and the email element with value "henk@hotmail.com" are given as input. We chose not to show the combination of the phone element with value "1111" with the roomnr element with value "1" in our example because they occur in the same possibility node. In that case our "pw" element only has one "pps" element.

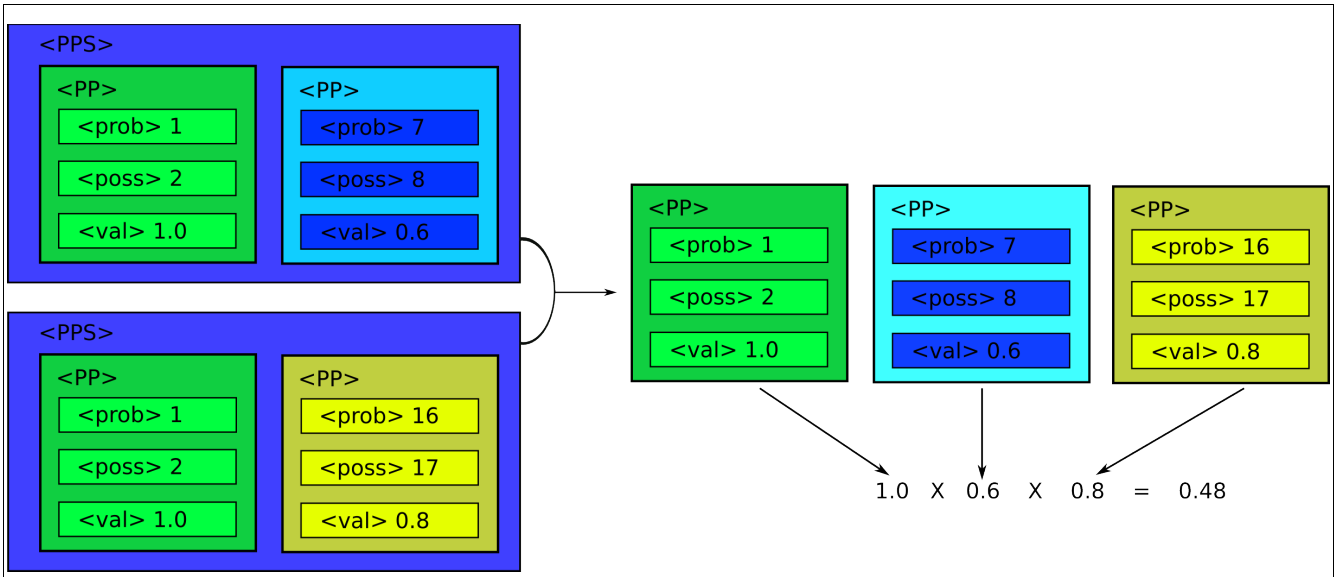


Figure 5.9: Calculating the final probabilities

The final probabilities are calculated according to the example of Figure 5.9: We take the "pps"-elements of a "pw" element and multiply the values of all distinct "pp" elements with each other. Doing so, double "pp" elements are only used once in this calculation, because as long as the path of ancestors of two elements is the same, following that path for one element automatically means following that path for the other node.

It might seem that the "createpossworlds" function is simple, because it only puts different "pps" nodes together in a "pw" element. This is true, but only for simple queries. For more complicated queries, the "createpossworlds" has a more important role to fulfill. To illustrate this we will now introduce a more complicated document (see Figure 5.10) on which we execute a more complicated query:

```
doc("figure5.10")/person[.//phone > "2222"]/info[.//roomnr<"3"]/mobilephone
```

```
(1)  <prob>
(2)     <poss prob="1">
(3)        <person>
(4)           <prob>
(5)              <poss prob="1">
(6)                 <info>
(7)                    <prob>
(8)                       <poss prob=".5">
(9)                          <roomdetails>
(10)                            <prob>
(11)                               <poss prob="0.4">
(12)                                 <roomnr>1</roomnr>
(13)                                 <phone>4444</phone>
                                   </poss>
(14)                               <poss prob="0.6">
(15)                                 <roomnr>4</roomnr>
(16)                                 <phone>2222</phone>
                                   </poss>
                                </prob>
                             </roomdetails>
                          </poss>
(17)                      <poss prob=".5">
(18)                         <roomdetails>
(19)                            <prob>
(20)                               <poss prob="0.2">
(21)                                 <roomnr>2</roomnr>
(22)                                 <phone>1111</phone>
                                   </poss>
(23)                               <poss prob="0.8">
(24)                                 <roomnr>3</roomnr>
(25)                                 <phone>3333</phone>
                                   </poss>
                                </prob>
                             </roomdetails>
                          </poss>
                       </prob>
(26)                   <prob>
(27)                      <poss prob="1">
(28)                         <mobilephone>0666</mobilephone>
                          </poss>
                       </prob>
                    </info>
                 </poss>
              </prob>
           </person>
        </poss>
     </prob>
```

Figure 5.10: More difficult example probabilistic XML document with node identifiers

The Java parser outcome for this query contains a double for-loop (see Figure 5.11), because we do not want to mix up results for "person" and "info" elements (although in this case this is not a major point because our document only has one "person" and one "info" element). The "computeprobs" function now gets a sequence of the result of three "getnids" function calls as argument. One for the candidate "mobilephone" elements, one for the predicate "phone" elements and one for the predicate "roomnr" elements.

```
import module namespace cpm = "http://monetdb.cwi.nl/XQuery/Documentation/Language/Modules/" at
"/path/to/cpm.xq";

<resultset>{
  for $i0 in
  doc("figure6.3.2.3")/prob/poss/child::person[.//phone > ”2222”]
  return
     for $i1 in $i0/prob/poss/child::info[.//roomnr < "3"]
     return
        cpm:computeprobs(
           (cpm:getnids($i1/prob/poss/child::mobilephone),
           cpm:getnids($i0//phone[.> ”2222”]),
           cpm:getnids($i1//roomnr[.< "3"]))
        )/result
}</resultset>
```

Figure 5.11: Xquery output of the Java parser.

The “computeprobs” function starts with comparing the probability and possibility nids of the mobilephone query with the nids of the phone query (see Figures 5.12 a and b).

```
<nids>
    <completenode>
        <node>
            <mobilephone>0666</mobilephone>
        </node>
        <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
            <pp><prob>26</prob><poss>27</poss><value>1.0<value></pp>
            <oavalue>1.0</oavalue>
        </pps>
        <subnodes/>
    </completenode>
</nids>
```

Figure 5.12a: The output of the “getnids” function for one mobilephone element.

```
<nids>
    <completenode>
        <node>
            <phone>4444</phone>
        </node>
        <pps>
            <pp><prob>1</prob><poss>2</poss>1.0<value/></pp>
            <pp><prob>4</prob><poss>5</poss>1.0<value/></pp>
            <pp><prob>7</prob><poss>8</poss>0.5<value/></pp>
            <pp><prob>10</prob><poss>11</poss>0.4<value/></pp>
            <oavalue>0.2</oavalue>
        </pps>
        <subnodes/>
    </completenode>
    <completenode>
        <node>
            <phone>3333</phone>
        </node>
        <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
            <pp><prob>7</prob><poss>17</poss><value>0.5<value></pp>
            <pp><prob>19</prob><poss>23</poss><value>0.8<value></pp>
            <oavalue>0.4</oavalue>
        </pps>
        <subnodes/>
    </completenode>
</nids>
```

Figure 5.12b: The output of the “getnids” function for two phone elements.

Since the "computeprobvals" function determines that both predicate elements (phone 4444 and phone 3333) are valid, the "createpossworlds" function should combine the nids of the mobilephone element with the nids both phone elements. However, the two phone elements cannot occur in the same world. In this situation, the more advanced functionality of the "createpossworlds" function is used. Based on the "pps" elements the "createpossworlds" function now creates two "pw" elements, one for each combination of the mobilephone element with one of the phone elements. This is shown in Figure 5.13:

```
<pw>
    <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
            <pp><prob>26</prob><poss>27</poss><value>1.0<value></pp>
            <oavalue>1.0</oavalue>
    </pps>
    <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
            <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
            <pp><prob>10</prob><poss>11</poss><value>0.4<value></pp>
            <oavalue>0.2</oavalue>
    </pps>
</pw>,
<pw>
    <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0</value></pp>
            <pp><prob>4</prob><poss>5</poss><value>1.0</value/></pp>
            <pp><prob>26</prob><poss>27</poss><value>1.0</value></pp>
            <oavalue>1.0</oavalue>
    </pps>
    <pps>
            <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
            <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
            <pp><prob>7</prob><poss>17</poss><value>0.5<value></pp>
            <pp><prob>19</prob><poss>23</poss><value>0.8<value></pp>
            <oavalue>0.4</oavalue>
    </pps>
</pw>
```

Figure 5.13: The output of the "createpossworlds" function.

The value of this division in possible world elements is that we can evaluate each possible world separately for any following predicate steps. We use the possible world elements as input for the "computeprobvals" function, in which we compare the "pps" elements with the "pps" elements of the new predicate elements: roomnr 1 and roomnr 2. We determine (using our rules) that roomnr 1 only occurs in the first possible world (with mobilephone 0666 and phone 4444), but that roomnr 2 occurs in neither possible worlds. Therefore the "computeprobvals" filters out the roomnr element with value "2" and the second "pw" element. The "createpossworlds" is called with the first "pw" element and the roomnr element with value "1". Because the roomnr element with value "1" and the phone element with value "4444" are positioned in the same possibility node no extra "pps" elements are added to the first "pw" element and the final result (before calculating the actual probability) is shown in Figure 5.14:

```
<pw>
    <pps>
          <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
          <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
          <pp><prob>26</prob><poss>27</poss><value>1.0<value></pp>
          <oavalue>1.0</oavalue>
    </pps>
    <pps>
          <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
          <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
          <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
          <pp><prob>10</prob><poss>11</poss><value>0.4<value></pp>
          <oavalue>0.2</oavalue>
    </pps>
</pw>
```

Figure 5.14: The output of the "createpossworlds" function for the final result.

The "computeprobs" function now takes the "node" element of the "getnids" result of the only mobilephone candidate element and returns this node with the calculated probability (1.0 x 1.0 x 1.0 x 0.5 x 0.4=0.2) obtained from the "createpossworlds" result.

5.2.5 Subnodes

In the prototype introduces thus far, a problem arises when we query for elements that contain children and we use this child element as predicate. An example is the following query executed on the document of Figure 5.10:

```
doc("figure5.10")/person/info/roomdetails[./roomnr="1"]
```

The "computeprobs" function would notice that there is one roomdetails element and one roomnr element that satisfy this query. So the probability of the roomdetails element (0.5) will be multiplied with the probability of the roomnr element (0.4) and the whole roomdetails element will be returned with a result probability of 0.2 as shown in Figure 5.15:

```
<resultset>
     <result val="0.2">
        <roomdetails>
           <prob>
              <poss prob="0.4">
                 <roomnr>1</roomnr>
                 <phone>4444</phone>
              </poss>
              <poss prob="0.6">
                 <roomnr>4</roomnr>
                 <phone>2222</phone>
              </poss>
           </prob>
        </roomdetails>
     </result>
</resultset>
```

Figure 5.15: The output of the "computeprobs" function for the final incorrect result.

Returning the whole roomdetails element with its entire subtree is not correct because it returns two worlds: one in which roomdetails contains a roomnr "1" and a phone "4444" and one in which roomdetails contains a roomnr "4" and a phone "2222". However, this second world is not part of our query result, since we asked only for roomdetails elements that contain a roomnr equal to "1". To prevent that child nodes of the result are listed while they cannot occur in the same world as the predicate elements of the query, we do an extra check for all descendants of the result elements.
To be able to check child elements the "getnids" result for an element containing child elements is extended as shown in Figure 5.17. We fill the "subnodes" element with the "completenode" elements that correspond with the children of roomdetails. Instead of returning the roomdetails element, we compare the "pps" elements of all child elements with the "pw" elements returned by the "createpossworlds" function. When the "pps" element of the child can be added to a current "pw" element, the child belongs to the result. If a new "pw" elements needs to be constructed when we would add the "pps" element of the child to the "pw" elements, we can conclude that the child element doesn't occur in the same possible world as the predicate element.
We construct a new roomdetails elements that contains only those child elements that occur in the same possible world as the predicate (or one of the predicates). We also make sure the the possibility nodes of the predicate elements itself gets the probability value "1.0". The new and correct result is shown in Figure 5.16:

```
<resultset>
     <result val="0.2">
        <roomdetails>
           <prob>
              <poss prob="1.0">
                 <roomnr>1</roomnr>
                 <phone>4444</phone>
              </poss>
           </prob>
        </roomdetails>
     </result>
</resultset>
```

Figure 5.16: The output of the "computeprobs" function for the final correct result.

```
<nids>
   <completenode>
      <node>
         <roomdetails>
            <prob>
               <poss prob=”0.4”>
                  <roomnr>1</roomnr>
                  <phone>4444</phone>
               </poss>
               <poss prob=”0.6”>
                  <roomnr>4</roomnr>
                  <phone>2222</phone>
               </poss>
            </prob>
         </roomdetails>
      </node>
      <pps>
         <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
         <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
         <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
         <oavalue>0.5</oavalue>
      </pps>
      <subnodes>
         <completenode>
            <node><roomnr>1</roomnr></node>
            <pps>
               <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
               <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
               <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
               <pp><prob>10</prob><poss>11</poss><value>0.4<value></pp>
               <oavalue>0.2</oavalue>
            </pps>
            <subnodes/>
         </completenode>
         <completenode>
            <node><phone>4444</phone></node>
            <pps>
               <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
               <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
               <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
               <pp><prob>10</prob><poss>11</poss><value>0.4<value></pp>
               <oavalue>0.2</oavalue>
            </pps>
            <subnodes/>
         </completenode>
         <completenode>
            <node><roomnr>4</roomnr></node>
            <pps>
               <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
               <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
               <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
               <pp><prob>10</prob><poss>14</poss><value>0.6<value></pp>
               <oavalue>0.3</oavalue>
            </pps>
            <subnodes/>
         </completenode>
         <completenode>
            <node><roomnr>2222</roomnr></node>
            <pps>
               <pp><prob>1</prob><poss>2</poss><value>1.0<value></pp>
               <pp><prob>4</prob><poss>5</poss><value>1.0<value></pp>
               <pp><prob>7</prob><poss>8</poss><value>0.5<value></pp>
               <pp><prob>10</prob><poss>14</poss><value>0.6<value></pp>
               <oavalue>0.3</oavalue>
            </pps>
            <subnodes/>
         </completenode>
      </subnodes>
   </completenode>
</nids>
```

Figure 5.17: The output of the “getnids” function one roomdetails element.

5.3 Observations

The most important observations using the compare paths method can be summarized as follows:

- When output elements are returned that contain a lot of child elements, for all these elements a check is performed to examine whether they are valid for the possible world in which the parent element occurs. This process is inefficient for the following two reasons:
  - As shown in Figure 5.17 the "getnids" function returns for such a parent element the nids of both the element itself and the nids for all his children. For the example this means that the "pp" elements with prob-poss combinations (1,2)(4,5) and (7,8) are listed in total five times, although it is obvious that the children all contain the "pp" elements of their parent.
  - The depth of the predicate is not taken into account when checking. So all descendant elements are checked for validity although the predicate itself might be a direct child element of the result element.
- Using the possibility per node representation style, no combinations of results elements can be reconstructed from the result. In other words; we cannot give the probability that two of the elements in the result occur in the same possible world.
- The size of the result grows linearly with the number of elements that satisfy the query.
- Calculations are done only for candidate elements i.e. those elements that have a chance to be in the final result (based on the original query, that is converted to a probabilistic variant).

```
<prob>
  <poss prob="1">
    <person>
      <prob>
        <poss prob="1">
          <firstname>Eliza</firstname>
        </poss>
      </prob>
      <prob>
        <poss prob="1">
          <lastname>David</lastname>
        </poss>
      </prob>
      <prob>
        <poss prob="1">
          <phones>
            <prob>
              <poss prob="1">
                <homephone>057-5049214</homephone>
              </poss>
            </prob>
            <prob>
              <poss prob="0.34">
                <mobilephone>06-18624061</mobilephone>
              </poss>
              <poss prob="0.66">
                <mobilephone>06-85489600</mobilephone>
              </poss>
            </prob>
          </phones>
        </poss>
      </prob>
      <prob>
        <poss prob="1">
          <address>
            <prob>
              <poss prob="1">
                <street>Madridweg</street>
              </poss>
            </prob>
            <prob>
              <poss prob="0.02">
                <housenr>576</housenr>
              </poss>
              <poss prob="0.98">
                <housenr>111</housenr>
              </poss>
            </prob>
            <prob>
              <poss prob="1">
                <postalcode>8244HV</postalcode>
              </poss>
            </prob>
            <prob>
              <poss prob="1">
                <city>Luinjeberd</city>
              </poss>
            </prob>
          </address>
        </poss>
      </prob>
      <prob>
        <poss prob="1">
          <emails>
            <prob>
              <poss prob="1">
                <email>Eliza.David@hotmail.com</email>
              </poss>
            </prob>
          </emails>
        </poss>
      </prob>
    </person>
  </poss>
</prob>
```

Figure 6.1: An example of one person in a generated test-addressbook-document

6 Experiments

6.1 Experimental set-up

To compare the efficiency of the different methods described in this report we setup a test scenario. We generate sample XML files as input for the different prototypes. The sample XML files all describe an uncertain address book, although the number of persons in the address book and the level of uncertainty differs. We create address books with the following combinations of properties:

Table 6.1: Combinations for generated test address books with the number of possible worlds they contain.

Number of PWs	Layout-code 2-1-0%	3-1-0%	3-2-0%	3-2-20%	3-2-50%
Number of persons					
5	2.09e+6	1.19e+12	9.52e+12	4.04e+11	2.14e+13
10	1.09e+12	1.38e+22	2.21e+25	9.25e+26	2.21e+30
20	1.98e+28	2.03e+46	3.33e+49	2.39e+51	1.36e+63
100	1.08e+127	1.24e+223	1.38e+224	9.07e+254	2.58e+281
1000	Inf	Inf	Inf	Inf	Inf
10000	Inf	Inf	Inf	Inf	Inf

In the code *a-b-c%* of Table 6.1, *a*, *b* and *c* are explained as follows:

a: leaf-nodes (firstname, lastname, homephone, mobilephone, street, housenr, postalcode, city and email) have a maximum of ***a*** possibilities
b: leaf nodes which can occur multiple times (homephone, mobilephone and email) have a maximum of ***b*** occurrences.
c: ***c*** percent of the parent nodes (phones, address email) has a maximum of 2 possibilities (the other part has one possibility).

The person elements and the address element that these persons contains have both a probability of one.

There are a few main cases in which our different prototypes may achieve a different level of performance. These are (together with the queries we specified to test the cases):

- Selecting all leaf nodes (like homephone or city)
  Q1 /person/phones/homephone
- Selecting all parent nodes (like person, phones or address)
  Q2 /person/phones
- Selecting a leaf node that has a specific value
  Q3 /person/phones/homephone[.=*X*]
- Selecting a parent node that has a certain leaf node or a leaf node with a specific

value.
Q4 /person/phones[./homephone]
Q5 /person/phones[./homephone=*X*]
Q6 /person[./phones/homephone=*X*]
Where *X* is an existing homephone number in the document that is queried.

We will describe the results of the queries ran by the different prototypes over our test files in the next section. It has to be mentioned that the prototypes we tested for the recursive path analysis and the compare paths method both support only one representation style. The recursive path analysis uses the possible-world style, while the compare paths method uses the possibility-per-node style. The system we use to run our prototypes has the following specifications:

- Pentium 4 2.66GHz
- 1GB ram
- Suse Linux 10.0 kernel 2.6.13-15.18-default
- MonetDB Server v4.22.1
- MonetDB/XQuery module v0.22.1

6.2 Results

First of all we tested the naive method. Its outcomes should have formed the basis for the comparisons done with the other two prototypes. However, we do not have any usable test data for the naive method, because every test case caused a crash of the MonetDB/XQuery database. The reason for this was the huge number of possible worlds that needed to be calculated. For example, our most simple test file with five persons consists of 2,097,152 different possible worlds. Since in our opinion, this sufficiently proved the inefficiency of the naive method, we continued with the tests for the other prototypes to examine whether they would outperform the naive method and which of both would turn out to be the most efficient method.

We started testing both remaining methods (CPM and RPA) using a small dataset, namely the 5-person address books in the different layouts we mentioned in Table 6.1. This first query (Q1) doesn't contain a predicate and the generated results do not contain any subnodes.

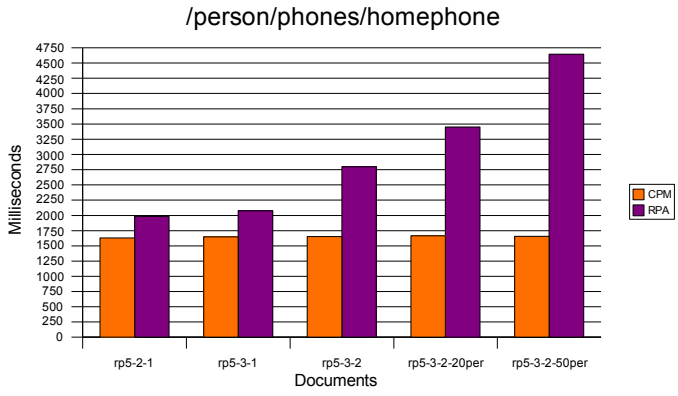


Figure 6.2: Testresults for Q1 with 5 persons

Figure 6.2 shows the time (in milliseconds) it takes to get all homephone elements from our dataset. It is clear that the processing time of the compare paths method stays constant around 1600 milliseconds. The processing time of the recursive path analysis increases up to 4600 milliseconds as the level of uncertainty grows, since the number of possible answers that need to be generated increases.
Next, we did the same test but time we queried all phones elements. These

phones elements contain subnodes: they are the parents of the homephone elements and are positioned one level higher in the XML tree.

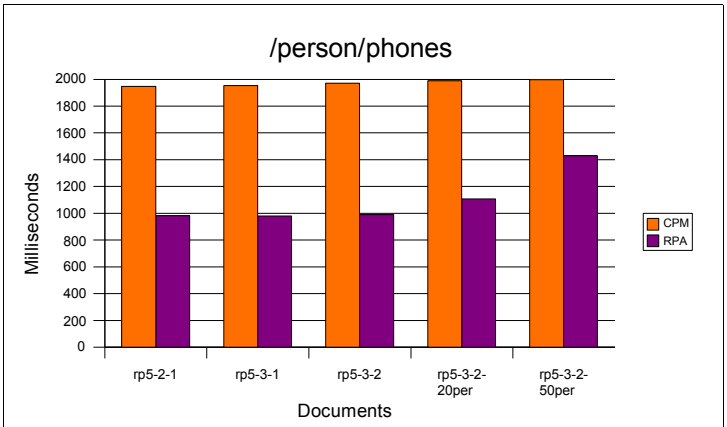


Figure 6.3: Testresults Q2 with 5 persons

In Figure 6.3 the test results for both methods are displayed. For the compare paths method the fact that in this test, the getnids function generates nids for both phones and its subnodes, causes a longer processing time (of around 2000 milliseconds).
The processing time for the recursive path analysis decreases to less than one second for the test documents rp5-2-1 to rp5-3-2, which don't contain uncertainty on the phones level. For the documents that contain 20 or 50 percent uncertainty on the phones level, the processing time increases slightly.

For both Q1 and Q2 it can be concluded that the CPM processing time is independent of the level of uncertainty. In contrast, the processing time of the RPA increases when the level of uncertainty rises. We can explain this difference by taking a closer look at the idea behind the both prototypes. The compare paths method compares paths to examine if nodes are valid (see also below, when executing queries containing predicates is discussed). However, since the queries we used in Q1 and Q2 do not contain predicates, each node is valid. That is why in CPM no checking is performed in all of the cases. The recursive path analysis constructs for each step of the query all possible worlds for nodes that satisfy that step. In case of uncertainty on the phone level, the number of possible worlds increases considerably. This causes the increasing processing time for Q1. On the other hand when having no uncertainty, no possible worlds have to be calculated, which explains the processing time less than a second for Q2.

In the next test, the influence of a predicate in the query in combination with subnodes in the result was investigated. In Q4, all phones that contain a homephone element were selected. In Figure 6.4, the results of Q4 are represented.

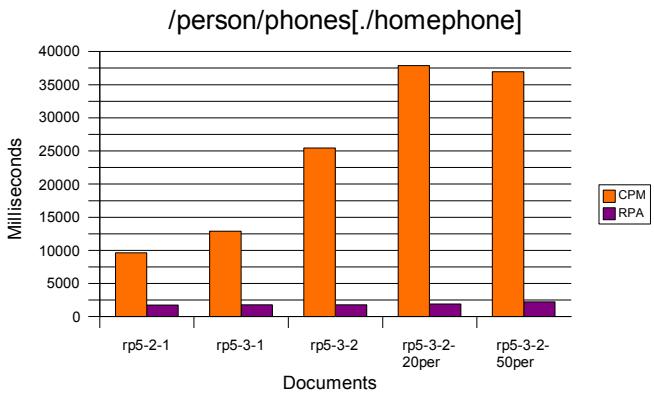


Figure 6.4: Testresults for Q4 with 5 persons

As can be seen, CPM's performance decreases drastically to 37.5 seconds for a document with uncertainty on phones level. The recursive path analysis handles the same queries in less then 2.5 seconds.

In Q6, instead of descendants of a person, a total person with a particular homephone is returned in the result. Test results are depicted in Figure 6.5.

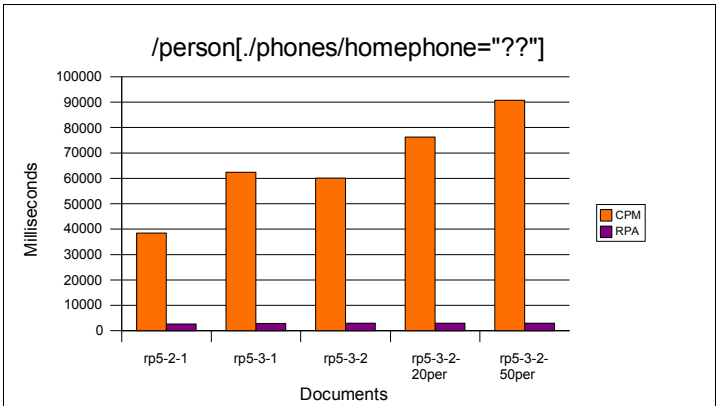


Figure 6.5: Testresults for Q6 with 5 persons

The differences between both query evaluating methods are even more pronounced than in the results of Q4: more than 1.5 minute is needed to get one person using the compare path method, whilst 3 seconds is enough for the recursive path analysis.
These differences can be explained as follows:
When executing Q6 we query for elements that contain child nodes. It is possible that some children are not valid for the result (as explained in 5.2.5). CPM performs a check to prevent the occurrence of invalid children in the result, whilst RPA does not. Therefore, the comparison between both methods is not completely fair for this kind of query. Thus, Figure 6.5, in which such an comparison is shown, should be seen as an indication of the processing time needed for both methods, instead of a proof that the recursive path analysis performs better in this situation.

The increased CPM processing times for both Q4 and Q6 are caused by the fact that both contain a predicate. In the evaluations of these queries, for each subnode it is checked whether it is valid (as explained in section 5.2.5), which is a rather inefficient procedure. The number of subnodes influences the processing time. This is why it takes longer to query bigger nodes (like person in Q6) as well as to execute a query on a document with more uncertainty (like the 50% documents in Figure 6.4 and Figure 6.5).

In the previous tests we have seen that the compare paths method can execute queries without leaf elements in less than 2 seconds independent of the level of uncertainty. To test the boundaries of this method in terms of the size of the dataset, we repeated Q1 using CPM with address books that contained larger numbers of persons (see Figure 6.6).

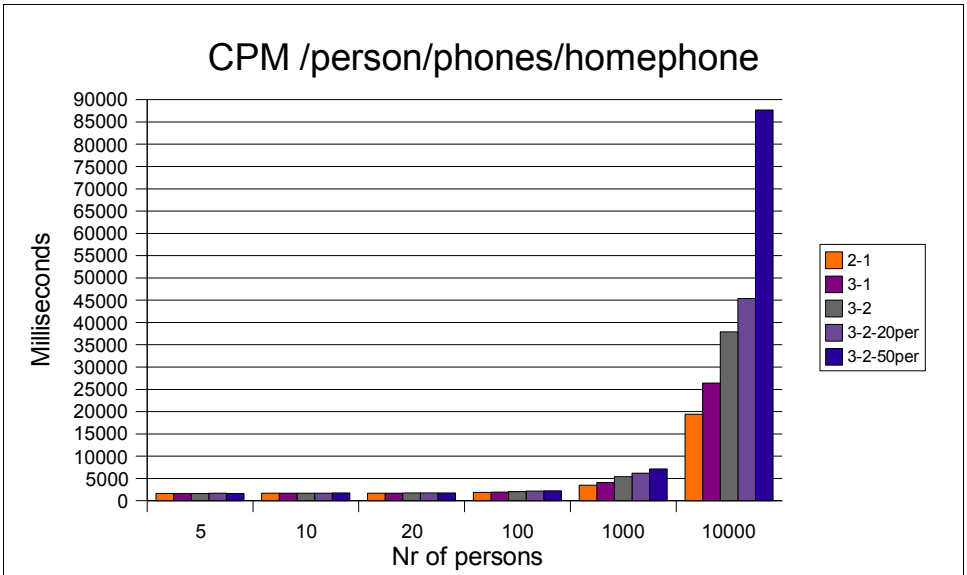


Figure 6.6: Testresults for Q1 executed on addressbooks of different sizes using CPM

This figure shows that even for a 100-person address book, the processing time increases barely. When querying address books with 1000 or 10000 persons the process slows down. This is probably the logical effect of the fact that large output has to be generated together with a large amount of probabilities that have to be calculated.

We performed Q1 with large address books again using RPA. The results of this query are shown in figure 6.7.

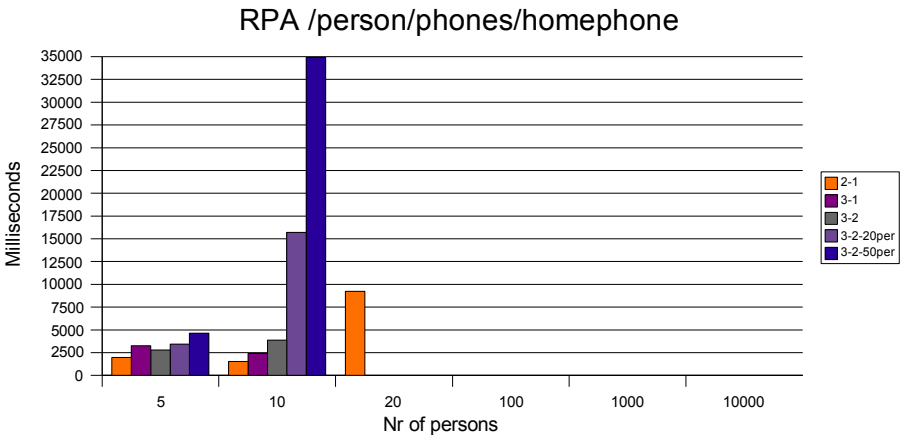


Figure 6.7: Testresults for Q1 executed on addressbooks of different sizes using RPA

The large number of possible worlds that have to be constructed leads to the fact that, except from the address book in its most simple form (2-1), address books with 20 or more persons can not be queried.
Finally, we chose to take a closer look on the CPM because of the promising results that were obtained in Q1 for different sizes of address books. We assumed that the long processing time in the query of the largest address books could be assigned to the large amount of results that had to be returned. To investigate if CPM could be useful for selecting only a few elements from a large dataset, we tested two more queries.

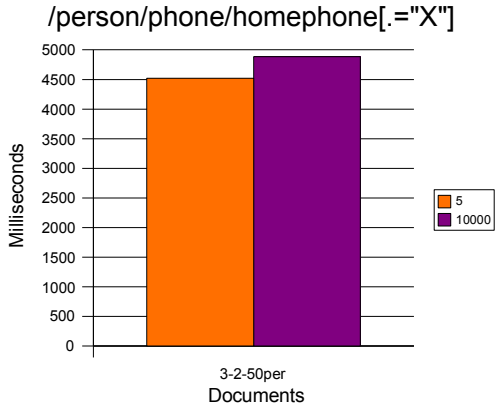


Figure 6.8: Testresults for Q3 executed on different sizes of addressbooks using CPM

First, we queried only for a homephone with a certain value (Q3, see Figure 6.8). Thus, in Q3, as in Q1, we query leaf nodes. However, Q3 returns only one element, since it contains a predicate that is fulfilled by one specific homephone element existing in the document that is queried. Because no lower subnodes are present, no checking for valid subnodes is performed. For this reason, such a query can be executed on the 5-persons address book in less than five seconds. The observation that the 10000-person address book can be queried in approximately the same amount of time, results from the fact that the selection as well as the representation processes of only element can be performed in a small amount of time.

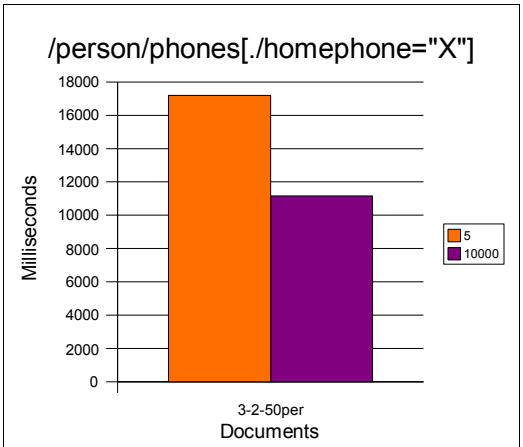


Figure 6.9: Testresults for Q5 executed on different sizes of addressbooks using CPM

In our final query (Q5), we queried all phones that contain a homephone with a specific value. Thus, contrary to Q3, Q5 does contain subnodes that are checked for validity. We chose the value for the homephone element in such a way that this query returns exactly one phones element. Running this query on the most complicated variant (3-2-50%) of the 5-person address book takes 17 seconds, but running it on the same variant of the 10000-address book only takes 11 seconds (Figure 6.9). The fact that in some cases it is possible to achieve lower processing times for larger address books than for small ones can be explained as follows: because the test files are randomly generated, it can be the case that the phones element requested in the small address book contains more subnodes than the one requested in the larger address book. Thus, the present results can be considered a coincidence.

6.3 Test conclusions

- The large number of possible worlds that are generated by the naive method, causes that none of our data-sets can be queried. So it can be concluded that the efficiency of this method is insufficient for practical use.
- The processing time of a query using the recursive path analysis varies with the level of uncertainty in the result. A high level of uncertainty causes long processing times. This makes that the recursive path analysis works well on a broad range of documents and queries, until the number of possible worlds in the result gets too high.
- The processing time of the compare paths method is almost independent of the level of uncertainty of the document. Therefore, queries for elements that do not contain subnodes, can be executed in a small amount of time.
- However, checking for valid subnodes in the compare paths method is such a slow process that querying for nodes with a large number of subnodes in combination with a predicate is no option for almost all documents. The recursive path analysis however only supports queries with predicates, if the query contains a following step in which the element does not occur in the same probability node as the predicate element.
- When the results itself is small sequence with a each small number of subnodes, the compare paths method can be used on very large (10000-person) files.

Based on these conclusions we can give a decision tree as shown in Figure 6.10. A user that has a certain knowledge about the document and what his query probably would return, can use this tree to choose a suitable query method.

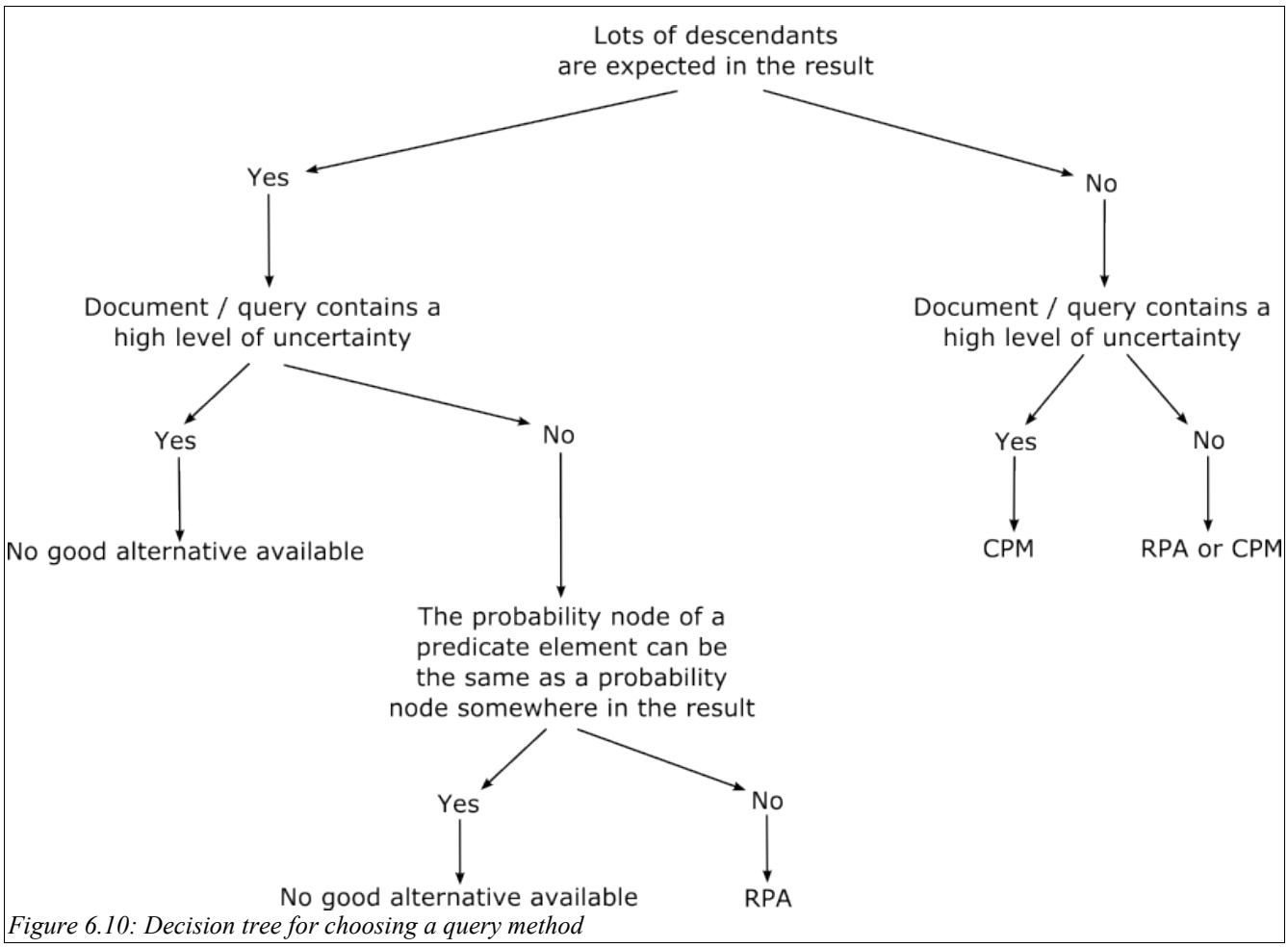


Figure 6.10: Decision tree for choosing a query method

7 Conclusions & recommendations

In this thesis, we have presented three methods for querying XML directly on the compact representation. In the naive method, all possible worlds are constructed and the query is executed over all these worlds. We have shown that the naive method is unsatisfactory for even a small dataset. We then described and tested two new methods for querying the compact representation: The "Recursive path analysis" and the "Compare paths method". We found out that both methods outperform the naive method; however, there are differences in performance between the two methods. Each method has its own strong and weak points.

The recursive path analysis method generates an XQuery that selectively considers possible worlds that are relevant to the query. A strong point of the recursive path analysis method is its ability to produce an answer in the possible worlds representation style. The improvement compared with the naive method is achieved by avoiding most of the possible world construction. The weak point of the recursive path analysis is the fact that for every step taken in the XPath query all possible worlds for the nodes satisfying that step are enumerated. Doing "deep" queries on documents with a lot of uncertainty still causes too many possible worlds to be enumerated.

The compare paths method compares the paths of nids of candidate elements with those of predicate elements to determine whether candidate elements should be included in the final answer. A strong point of this method is its use of the possibility-per-node answer representation. Moreover, for several types of queries the processing time of the compare paths method is independent of the level of uncertainty or the size of the document. This makes it possible to query an address book document containing up to 10000 people in roughly the same amount of time as a five person address book. An disadvantage of the compare paths method concerns the filtering of the result subtrees. If the query result contains elements with a lot of child elements, the response time will be long.

The recursive path analysis uses the possible world representation style. Although the result is very precise this way, it generates very large results. If we do not take this issue into account we can conclude the following about the two methods:

- Using small documents (10 pers.) or larger documents containing minimal uncertainty (1000 pers.), overall, the recursive path analysis performs best.
- The compare paths method performs better only if the elements that are queried have (almost) no child elements. If this is the case, even very large probabilistic XML documents (10000 pers.) can be queried with the same speed as a small document (5 pers.).

The execution times of both methods are not directly comparable. The compare paths method filters the subtrees of result elements. The recursive path analysis doesn't do so. For this reason, using the recursive path analysis, correct results are not guaranteed when queries containing a predicate are executed.

In conclusion, the compare paths method has the most potential to form the basis for an efficient probabilistic XML query mechanism. The result subtree filtering technique

deserves further research as it currently appears to be one of the major bottlenecks.

Because of the different answer representation styles the two prototypes use, our comparison is not completely fair. The performance of the recursive path analysis decreases when documents containing more uncertainty are queried. In the same situation the performance of the compare paths method stays roughly the same. However, the current investigation doesn't answer the question whether this difference in performance is caused by a fundamental difference between the algorithms of both prototypes or by the considerable difference between the sizes of the results that need to be generated. An additional experiment comparing both prototypes using both the possibility-per-node style and the possible world style could give a better view on this issue.

7.1 Optimization recommendations

As mentioned before, the processing time of the compare paths method increases rapidly when returning elements with large subtrees. The method to check whether child elements may occur in the result needs to be optimized to get a prototype that is more usable in practice. Possible solution directions are:

- The number of subnodes that are checked for validity should be reduced to a minimum. In many cases it is not necessary to check all descendants. For predicate elements it is not needed at all.
- For the structure of a path of prob and poss nids, a way of efficiently representing subnodes should be investigated, so that less redundant data is needed.
- The function that takes care of returning paths of prob and poss nids should have a different output for candidate and predicate elements. Furthermore it isn't necessary to call this function when no predicates are involved in the query.

Another way of optimizing the compare paths method can be done by implementing the the function that compares paths and the “createpossworlds” function in the internal programming language of MonetDB as a standard function instead of an XQuery module.

7.2 Extension recommendations

The compare paths method prototype supports the most general XPath queries. More advances applications of XPath are not yet (fully) implemented. Axis steps are for example already supported in the Java parser but whether the XQuery module can handle them is not yet fully tested. Some operators (like “+”, “-”, “<=” , “>=”, “eq” and “or” for example) and functions (like “position()” and “last()” for example) however shouldn't be a problem for the XQuery module but are not yet supported in the Java parser.

When the goal is to fully support XQuery not only the parser has to be extended, also some important questions have to be asked. The most used feature of XQuery is in general to iterate over XPath query results. But when we have probabilistic results, where do we want to iterate over? different worlds? result elements? combinations of result elements? And what do we do with the probabilities? Do we want to use them for further calculation or ignore them? Do the probabilities differ when we specify a where clause in our for loop?
We do not have the answers on these questions but hopefully they form a challenge for future research.

References

[AKO07] L. Antova, C. Koch, and D. Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *ICDE'07*, pages 1479–1480.

[BGK+06] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery - Fast and Scalable XQuery Processor Powered by a Relational Engine. In *ACM SIGMOD International Conference on Management of Data*, Chicago, USA, June 26-29, 2006, June 2006.

[CD99] J. Clark and S. DeRose. XML path language (Xpath) 1.0. W3C recommendation. World Wide Web Consortium, `http://www.w3.org/TR/xpath`, Nov. 1999.

[CFR+00] D.Chamberlin, D.Florescu, J.Robie, J.Simeon, and M.Stefanescu. XQuery: A query language for XML. World Wide Web Consortium, `http://www.w3.org/TR/xquery`, Feb 2000.

[CFP00] S. Ceri, P. Fraternali, S. Paraboschi. XML: Current Developments and Future Challenges for the Database Community. In *Proc. of the 7th Int. Conf. on Extending Database Technology (EDBT)*, Springer, LNCS 1777, Konstanz, March, 2000.

[DAF04] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML documents in relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of Data (SIGMOD'99)* , Philadelphia, Pennsylvania, May 31 - June 3 1999.

[FK99] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *IEEE Data Engineering Bulletin*, 22(3), September 1999.

[HGS03] Edward Hung, Lise Getoor, and V.S. Subrahmanian. Probabilistic interval XML. In *International Conference on Database Theory (ICDT)*, Siena, Italy, January 2003.

[JAC+02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. In *The VLDB Journal*, pages 274–291, 2002.

[KEI06] A. de Keijzer. Probabilistic XML in Information Integration. In *Proceedings of the VLDB2006 Ph.D. Workshop*, Seoul, Rep of Korea, 2006.

[KEU08] M. van Keulen, Personal communication, March 2008.

[KK04] A. de Keijzer and M. van Keulen. A possible world approach to uncertain relational data. In *Proc. SIUFDB Workshop*, Zaragoza, Spain, Sept. 2004.

[KKA05] M. van Keulen, A. de Keijzer, and W. Alink. A probabilistic xml approach to data integration. In *Proc. ICDE Conf.*, Tokyo, Japan, pages 459–470, 2005.

[KKL06] A. de Keijzer, M. van Keulen, and Y. Li. Taming Data Explosion in Probabilistic Information Integration. In *Proceedings of the International Workshop on Inconsistency and Incompleteness in Databases (IIDB)*, March 26, 2006, Munich, Germany, March 2006.

[KKR+00] G. Kappel, E. Kapsammer, S. Rausch-Schott, W. Retschitzegger: X-Ray - Towards Integrating XML and Relational Database Systems. In *Conceptual Modeling - ER 2000 - 19th Int. Conference on Conceptual Modeling*, Salt Lake City, Utah, USA, October 9-12, 2000, LNCS 1920, Springer-Verlag, Berlin, 2000, pages 339–353.

[NJ02] A. Nierman and H.V. Jagadish. ProTDB: Probabilistic data in XML. In *Proc. of the 28th VLDB Conference*, Hong Kong, China, August 2002.

[WID05] J. Widom, "Trio: A System for Integrated Management of Data, Accuracy, and Lineage," In *CIDR*, 2005.