Journal of Universal Computer Science, vol. 12, no. 11 (2006), 1651-1678
submitted: 1/5/06, accepted: 15/10/06, appeared: 28/11/06 © *J.UCS*

Magic Sets for the XPath Language

Jesús M. Almendros-Jiménez
(Dept. de Lenguajes y Computación Universidad de Almería, Spain
jalmen@ual.es)

Antonio Becerra-Terón
(Dept. de Lenguajes y Computación Universidad de Almería, Spain
abecerra@ual.es)

Francisco J. Enciso-Baños
(Dept. de Lenguajes y Computación Universidad de Almería, Spain
fjenciso@ual.es)

Abstract: The eXtensible Markup Language (XML) is considered as the format of choice for the exchange of information among various applications on the Internet. Since XML is emerging as a standard for data exchange, it is natural that queries among applications should be expressed as queries against data in XML format. This use gives rise to a requirement for a query language expressly designed for XML resources. World Wide Web Consortium (W3C) convened to create the XQuery language, concretely, a typed functional language for querying XML documents. One key aspect of the XQuery language is the use of the XPath language as basis for handling the structure of an XML document. In this paper, we present a proposal for the representation of XML documents by means of a logic program. Rules and facts can be used for representing the document schema and the XML document itself. In addition, we study how to query by means of the XPath language against a logic program representing an XML document. It evolves the specialization of the logic program with regard to the XPath expression. This specialization technique is based on the well-known transformation technique called Magic Sets and studied for deductive databases. The bottom-up evaluation of the specialized program is used for answering the query in the XPath language.
Key Words: Logic programming, XPath language, Magic sets
Category: D.1.6, H.2.3

1 Introduction

The eXtensible Markup Language (XML) is considered as the format of choice for the exchange of information among various applications on the Internet. The use of tags makes XML data self-describing, and the extensible nature of XML makes it possible to define new kinds of documents for specialized purposes. As the importance of XML has increased, a number of standards has grown up around it, many of which were defined by the *World Wide Web Consortium* (W3C). For example, *XML Schema* [W3C01] provides a notation for defining new types of elements and documents; *XML Path Language* (XPath) [W3C04c] provides a notation for selecting elements within an XML document; and finally, *eXtensible StyleSheet Language Transformations* (XSLT) [W3C04e] provides a notation for

transforming XML documents from one representation to another. Since XML is emerging as a standard for data exchange, it is natural that queries among applications should be expressed as queries against data in XML format. This use gives rise to a requirement for a query language expressly designed for XML resources. W3C convened to create the *XQuery language*, concretely, a typed functional language for querying XML documents [W3C04d, Wad02, Cha02]. One key aspect of the XQuery language is the use of the language XPath as basis for handling the structure of an XML document.

Logic-based languages have been proved useful in many areas since they allow to build small, declarative and extensible programs. For the database area, for instance *Datalog* has been investigated for querying and rule-based data manipulation. One of the key aspects of database languages based on logic programming, Datalog among others, is the use of a *fixed-point operator* [Apt90] as evaluation mechanism, following a *bottom-up evaluation* of the program for query solving. For efficiency reasons, a *program specialization* technique called *Magic Sets* [BR91] is achieved with regard to a goal.

A *XML document* basically is a *labelled tree* with nodes representing *composed or non-terminal items* and leaves representing *values or terminal items*. The *XML schema*, which is also an XML document, defines the structure of well-formed documents and thus it can be seen as a type definition. Therefore, well-formedness analysis can be seen as type checking [SW03, HP03]. XPath considered as query language expresses a query against an XML document. Essential to semi-structured data [ABS00] is the selection of data from incompletely specified data items as in an XML document. For such data selection, the XPath language is a path language which provides constructors similar to regular expressions and "wildcards" allowing a flexible node retrieval. For instance, let us consider the following XML document:

Example of XML Document

```
<books>
<book year="2003">
<author>Abiteboul</author>
<author>Buneman</author>
<author>Suciu</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
<review><em>The <em>best</em> ever!</em></review>
</book>
</books>
```

representing a set of books, where each record stores the authors, titles and reviews for each book; and each record has an attribute representing the publishing year. Now, with respect to the above XML document, we can consider

the following two XPath expressions, as well as the expected answers in XML format:

XPath Expression	Expected XML Answer
(1) `/books/book[author=Suciu]/title`	**(1)** `<title> Data on the Web </title>`
(2) `/books//title`	**(2)** `<title> Data on the Web </title>` `<title> XML in Scotland </title>`

where **(1)** requests `Suciu`'s book titles, and **(2)** requests book titles without taking into account the structure of the book records.

1.1 Contributions of the Paper

In this paper, we are interested in the use of logic programming for handling XML documents and XPath queries. In this context, our contributions can be summarized as follows:

- An XML document can be seen as a logic program, by considering *facts* and *rules* for expressing both the XML schema and document. On one hand, rules can describe the schema of an XML document in which a (possibly recursive) definition specifies the well-formed documents. On the other hand, each XML document can be described by means of facts, one for each terminal item.

- Our second contribution is that once XML documents can be described by means of a logic program, an XPath expression against the document requires to obtain a subset of the *Herbrand model* [Apt90] represented by the logic program. In deductive databases, the *bottom-up-based computation model* [BR91, ABS01] specializes a logic program with regard to a given query in order to compute the *subset of the Herbrand model* needed for answering the query. Our idea is to provide a *specialization program method*, but in this case, for handling of XPath expressions. Therefore, we will specialize the logic program representing an XML document with regard to an XPath expression in order to get the answer; that is, the XML data relevant to the query. This specialization technique is based on *Magic Sets* technique allowing a *bottom-up evaluation* of the specialized program in order to obtain the answer of the query.

  Although the XML schema is usually available for XML documents, our method has been studied for extracting the XML schema from the XML document itself. It can be considered in a certain sense as type inference. However, we think that we might adapt our technique to directly translate XML schemas (or DTD's) into logic rules.

- Our technique allows the handling of XML documents as follows. Firstly, the XML document is loaded. It involves the translation of the XML document

into a logic program. For efficiency reasons, the rules corresponding to the XML schema are loaded in *main memory*, but facts, which basically represent the XML document, are stored in *secondary memory* (using appropriate *indexing techniques*). Secondly, the user can now write queries against the loaded document. For query solving, the logic program (corresponding to the XML schema) is specialized for each query, and the bottom-up evaluation of such specialized program computes the answer.

- We have developed a prototype called XINDALOG in which we have implemented *XPath* following the technique presented in this paper. It has been developed under SWI-Prolog and it is hosted at `http://indalog.ual.es/Xindalog`, from which it can be run. We will present *benchmarks* of our prototype w.r.t. not too structured XML documents and XML documents of big size.

1.2 Related Work

In order to handle XML documents, some logic languages and formalisms have been proposed. For instance, XCERPT [SB02] proposes a pattern and rule-based query language for XML documents using the so-called query terms including logic variables for the retrieval of XML elements. For this language, a specialized unification for query terms has been studied in [BS02]. The same can be said for XPathLog (LOPIX system) [May04], which is a Datalog-style extension of XPath with variable bindings. Elog [BFG01] is also a logic-based XML data manipulation language which has been used for representing Web documents by means of logic programming. This is also de case of X-Prolog [CF03] which can represent XML documents into logic programming by using the DTD definition. The Rule Markup Language (RULEML) [Bol01] translates Prolog facts and rules into XML documents allowing the combination of XML and RDF (Resource Description Framework) documents. Finally, some Prolog implementations include libraries for XML document loading and querying such as SWI-Prolog [Wie05] and CIAO [CH01]. In particular, the representation of XML documents using logic programs was proposed in PiLLoW [CH01]. In this proposal, the document or URL is represented as a fact and the document itself (i.e. XML or HTML code) as a Herbrand term, which the argument of this fact. In this way, the XML documents are represented as logic programs and also these logic programs are used in order to access, combine or generate the documents.

In some of cited approaches, [SB02, May04] XPath is directly handled, that is, rules and queries use a new kind of Prolog terms adapted to XML patterns. It involves to study new unification algorithms for the new Prolog terms. However, in our work we will show how to handle XML documents not introducing new Prolog terms, but using the traditional Prolog terms. It involves to define a

translation of the structure of XML documents to Prolog terms. It is the same case as Prolog implementations for loading XML documents using a unique Prolog term for the entire XML document. However, we think that our translation is more refined than Prolog implementations, and suitable for a more efficient evaluation method.

With respect to the XPath queries, in the approaches [SB02, May04], the XPath expressions are translated into Prolog goals using the new query terms, and in the Prolog implementations, XPath is handled by using Prolog predicates. In our case, the XPath queries involve a Magic sets-based program transformation and bottom-up evaluation. One of the advantages of the bottom-up evaluation is to obtain sets of facts in each step of evaluation, which is called *"set-at-time evaluation"* in contraposition to the *"tuple-at-time evaluation"* of the traditional top-down evaluation method. It is suitable for databases once facts can be stored in secondary memory and therefore *minimizing disk accesses*. The bottom-up evaluation of the transformed program will obtain the answer by means of the reconstruction of the XML document representing the result from the set of obtained facts. The reconstruction assumes the same criteria as the translation of XML document-logic program.

With regard to [Bol01], we translate XML documents into a logic program using facts and rules; however we are not still interested in the translation of logic rules into XML (or RDF) documents. This translation would be interesting when semantic information (for instance, ontologies) is handled by means of logic programming. In fact, our idea is to consider these aspects as future work. Our approach opens two promising research lines.

- The first one, the extending of XPath to a more powerful query language such as XQuery; that is, the study of the implementation of XQuery in logic programming. The current implementations of XQuery are implemented using as host language a functional language (XDUCE and GALAX) [HP03, MS03].
- The second one, the use of logic programming as inference engine for the so-called "Semantic Web", by introducing RDF documents or OWL (Ontology Web Language) [W3C04b, W3C04a].

1.3 Structure of the Paper

The structure of the paper is as follows. Section 2 will study the translation of XML documents into Prolog; Section 3 will present the indexing technique over XML documents represented by means of logic programming; Section 4 will discuss the magic-sets based technique applied to XPath queries; Section 5 will explain the combination of the indexing and magic sets techniques; Section 6 will show the prototype XINDALOG developed under SWI-Prolog for the language

XPath at the University of Almeria (url: http://indalog.ual.es/Xindalog); Section 7 will present a full set of XPath queries tested in our prototype; Section 8 will show a benchmark of XPath queries tested in our prototype, specifying their answer times with the program specialization and without program specialization w.r.t. an XPath query; and, finally, Section 9 will conclude and present future work.

2 Translating XML Documents into Prolog

In this section, we will show how to translate an XML document into a logic program. As commented in the introduction we will use a set of rules for describing the XML schema and a set of facts for storing the XML document. With this aim, we will show a set of general criteria for the building of the Prolog program from a XML document.

As running example, we will consider the following XML document representing a book database. In the XML document *tags* are used for specifying the structure of each XML element representing, in this case, a set of `books` described by means of the names of the authors, the title and a review. Each `book` is qualified by means of an *attribute* called `year`. For each *element* `book`, we have three grouped *sub-elements* `author`, `title` and `review`. In addition, the element `review` contains sub-elements used for formatting the text described by the review.

An Example of XML document:

```
<books>
<book year="2003">
<author>Abiteboul</author>
<author>Buneman</author>
<author>Suciu</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
<review><em>The <em>best</em> ever!</em></review>
</book>
</books>
```

Here, the XML database includes two books. The first one, edited in `2003`, with `authors` `Abiteboul`, `Buneman` and `Suciu`, and `title 'Data on the Web'`. Finally, the opinion of the reviewers for this book is: `A` *fine* `book`. The second one, edited in `2002`, was written by `Buneman` with title `XML in Scotland`, and the opinion of the reviewers is *The best ever!*. In this XML document, we can see typical features of a semi-structured data model [ABS00]: heterogeneous records, in particular, non-first normal relations, missing values, among others. Once shown the example, now, we focus on the general criteria for translating

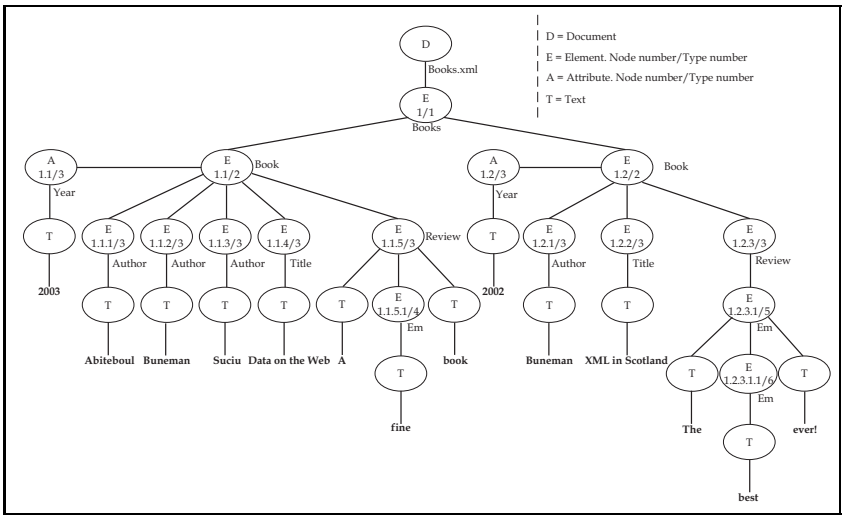

Figure 1: Node and type numbering in the tree structure of the above XML document

XML documents into a logic program. These criteria can be summarized as follows:

1. Each tag (element) is translated into a predicate name. Each predicate has *three arguments*.
   - The first one is used for building a *prolog term* containing the XML document.
   - The second argument of the predicate is used for *numbering each node* (called *node number*) of the XML document tree (see Figure 1).
   - The third one is use for *numbering each type* (called *type number*) (see Figure 1).

   Next, we will explain the use of node and type number in the logic program.
2. For un-tagged elements grouped together tagged elements the predicate name `unlabeled` is used.
3. Each non-terminal tag is translated into a function symbol named as "`name-element`" + `type` with an argument for each (sub)element and an additional argument for storing the list of attributes.
4. Each terminal element is translated into a fact, numbered as in the figure 1. For instance, the above XML document can be represented by means of a logic program as follows:

Prolog program of an XML document

```
Rules (Schema):                          Facts (Document):
---------------------------------------- ----------------------------------------
books(bookstype(A, []), L,1) :-          year('2003', [1, 1], 3).
       book(A, [B|L],2).                 author('Abiteboul', [1, 1, 1], 3).
book(booktype(A, B, C, [D]), L ,2) :-    author('Buneman', [2,1, 1], 3).
       author(A, [E|L],3),               author('Suciu', [3,1,1], 3).
       title(B, [F|L],3),                title('Data on the Web', [4, 1, 1], 3).
       review(C, [G|L],3),               unlabeled('A', [1, 5, 1, 1], 4).
       year(D, L,3).                     em('fine', [2, 5, 1, 1], 4).
review(reviewtype(A,B,[]),L,3):-         unlabeled('book.', [3, 5, 1, 1], 4).
       unlabeled(A,[J|L],4),             year('2002', [2, 1], 3).
       em(B,[K|L],4).                    author('Buneman', [1, 2, 1], 3).
review(reviewtype(A,[]),[I|L],3):-       author('Buneman', [1, 2, 1], 3).
       em(A,[J|L],5).                    title('XML in Scotland', [2, 2, 1], 3).
em(emtype(A,B,[]),L,5) :-                unlabeled('The', [1, 1, 3, 2, 1], 6).
       unlabeled(A,[G|L],6),             em('best', [2, 1, 3, 2, 1], 6).
       em(B, [H|L],6).                   unlabeled('ever!', [3, 1, 3, 2, 1], 6).
```

In this example, we can see the translation of each tag into a predicate name: `books`, `book`, etc. Each predicate has three arguments. The first one consists on a function symbol with the same name as the tag, but adding the suffix `type`, which encapsulates the XML document. Therefore, we have `bookstype`, `booktype`, etc. Each function symbol has arity $n + 1$ where n is the number of subelements of the given element, and it has an extra-argument which is devoted to store the attribute list. The second argument is used for numbering each node. We have numbered the nodes of the XML tree by levels and from *left to right* starting from 1. For instance, the three facts for the authors of the first book are numbered $[1, 1, 1]$, $[2, 1, 1]$ and $[3, 1, 1]$, representing the authors $'$`Abiteboul`$'$, $'$`Buneman`$'$ and $'$`Suciu`$'$, respectively, and $[1, 2, 1]$ for representing $'$`Buneman`$'$ in the second book (see Figure 1). Let us remark that the numbering in the facts is in reverse order with respect to the numbering in the XML tree due to the use of lists for representing them. The last argument of the predicates is a number used for numbering each type. It will explained in the next point. Finally, each element, which is not labelled but grouped, is translated into a label called `unlabeled`. This is the case of "`A`" and "`book`." in the first review.

5. In the case of a non-terminal does not always have the same structure, we introduce more than one rule with different type number for each kind of subelements. For instance, in the running example we have in each schema rule sequences of type numbers: `1-2`, `2-3-3-3-3`, `3-4-4`, `3-5`, `5-6-6`. From these ones, the most significative ones are `3-4-4` and `3-5` which distinguish two cases for the type `review`. The first type is used in the first book where `fine` is emphasized but "`A`" and "`book`." not. The second type is used in the second book where "`The best ever!`" is emphasized but "`best`" is doubly emphasized. The so-called *type number* is vital to distinguish each kind of type in the same element. Let us remark that the use of different type numbers for the

same "type" as `review` can be justified due to the occurrence of different instances of the same type in different positions in the document and possibly with some missing values. However, when the instances occur in the same position (i.e. level of the XML tree) and with the same kind of values, they are numbered with the same type number. To illustrate this problem let the following document be:

```
<books>
<book year="2003">
<author>Abiteboul</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
</book>
</books>
```

where we have two kinds of records, one with author, title, review and year, and the second one with author, title and year. In this case, we have to consider the following schema rules:

```
books(bookstype(A, []), L,1):-
        book(A, [C|L],2).
book(booktype(A, B, C, [D]), L,2) :-
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(D, L,3).
book(booktype(A, B, [C]), L,2) :-
        author(A, [G|L],4),
        title(B, [H|L],4),
        year(C, L,4).
author('Abiteboul',[1,1,1],3).
author('Buneman',[1,2,1],4).
...
```

The use of numbers **2-3-3-3-3** and **2-4-4-4** in the above rules and in the corresponding facts allows the distinction of the subelements of `Abiteboul` and `Buneman`'s books. The use of the same numbering can imply ambiguity, given that `Abiteboul`'s book is also of the type described by second rule of `book`. The same can be said for the case of `review` in the running example. The tag `review` has two rules, one for the case: "`A <em> fine <\em> book.`" and other one for " `<em> The <em> best <\em> ever! <\em>` ", where in the first case the sole emphasized text is "`fine`", and in the second case all is emphasized, and "`best`" is doubly emphasized. The facts and rules in this case are:

```
unlabeled('A', [1, 5, 1, 1], 4).
em('fine', [2, 5, 1, 1], 4).
unlabeled('book.', [3, 5, 1, 1], 4).
unlabeled('The', [1, 1, 3, 2, 1], 6).
em('best', [2, 1, 3, 2, 1], 6).
unlabeled('ever!', [3, 1, 3, 2, 1], 6).
review(reviewtype(A,B,[]),L, 3):-
       unlabeled(A,[J|L],4),
       em(B,[K|L],4).
review(reviewtype(A,[]),L,3):-
       em(A,[K|L],5).
em(emtype(A,B,[]), L,5) :-
       unlabeled(A,[G|L],6),
       em(B, [H|L],6).
```

They allow to distinguish that the first case is built from the first `review` rule and the second from the second `review` rule together with the `em` rule.

6. Whenever there is more than one value for the same sub-tag in the same element, we introduce one fact for each value, numbered with the same type number, but distinct node number. For instance w.r.t. the running example:

```
author('Abiteboul', [1, 1, 1], 3).
author('Buneman', [2, 1, 1], 3).
author('Suciu', [3, 1, 1], 3).
```

Let us remark that we could use a unique fact and use a Prolog list for storing the three elements. We believe that both choices can be sound, and our transformation technique could be adapted.

7. In the case of an element has several attributes, a Prolog list is used for storing them. For instance, with respect to the following document:

```
<book year="2003",keyword="XML">
<author>Abiteboul</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
```

we consider the following rule:

```
book(booktype(A, B, C, [D,J]), L, 2) :-
       author(A, [G|L],3),
       title(B, [H|L],3),
       review(C, [I|L],3),
       year(D,L,3),
       keyword(J,L,3).
```

8. In the case of recursively defined XML documents, they are handled by means of a recursive rule. For instance:

```
em(emtype(A,B,[]),L,5) :-
       unlabeled(A,[G|L],6),
       em(B, [H|L],6).
```

This rule expresses that an emphasized text can include other(s) emphasized text(s).

9. In the case of using the XML schema, XML types can be identified with Prolog-style types whenever it is possible. In the case of handling of *lists of integers* and *strings*, we introduce *lists*. *Integers*, *Reals*, etc can be handled by means of the type *Integer*, *Real* of the host language distribution. For instance, consider the following XML document and its corresponding rule:

```
<book year="1999 2003">
<author>Abiteboul</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
-----------------------------------------------
book(booktype(A, B, C, [Y]), L,2) :-
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(Y,L,3).
year(['1999','2003'],[1,1],1).
```

The use of XML types enables to use more sophisticated queries by using binary operators as $\geq, \leq, mod, ...$ and other string and list operations. For simplicity, in this paper we have only considered values in the type string, but other kinds of types can be considered as well as operations on them. Finally, let us remark that by using the XML schema we can adopt the name used by a XML "`complextype`" element for naming function symbols; and also the "`mixed`" attribute can be translated into "unlabeled" predicate name.

3 Indexing

In this section, we would like to present how to index XML documents represented by means of a logic program wherein the main goal is to obtain an efficient retrieval of XPath queries. It can be summarized as follows.

- We use *main memory* for the handling of schema rules.
- We use *secondary memory* for the handling of facts.
- We have considered *file indexing* for the facts.
- In particular, we have *two kind of indexes*: one for indexing *predicate names*, and other one for indexing each *group of items*.

The use of main memory for storing schema rules is justified due to in the most of cases the number of schema rules is small. The use of secondary memory

for storing facts is justified since XML documents can be too big to be stored in main memory. File indexing is justified for efficiency reasons. Firstly, our approach requires to recover facts for a given predicate, in this case we use the first kind of index. Secondly, our approach requires the recover of the elements grouped in the same record; in this case we use the second kind of index.

For instance, w.r.t. the running example, we need to generate the following set of indexes with respect this set of facts:

first index	second index
author	pos(1, 0).
	pos(2, 0).
	pos(3, 0).
	pos(9, 8).
em	pos(6,5).
	pos(12,11).

first index	second index
title	pos(4, 0).
	pos(10, 8).
unlabeled	pos(5, 5).
	pos(7, 5).
	pos(11, 11).
	pos(13, 11).
year	pos(0, 0).
	pos(8, 8).

group	facts
[1, 1]	(0) year('2003', [**1**, **1**], 3).
	(1) author('Abiteboul', [1, **1**, **1**], 3).
	(2) author('Buneman', [2, **1**, **1**], 3).
	(3) author('Suciu', [3, **1**, **1**], 3).
	(4) title('Data on the Web', [4, **1**, **1**], 3).
[5, 1, 1]	(5) unlabeled('A ', [1, **5**, **1**, **1**], 4).
	(6) em(fine, [2, **5**, **1**, **1**], 4).
	(7) unlabeled(' book.', [3, **5**, **1**, **1**], 4).
[2, 1]	(8) year('2002', [**2**, **1**], 3).
	(9) author('Buneman', [1, **2**, **1**], 3).
	(10) title('XML in Scotland', [2, **2**, **1**], 3).
[1, 3, 2, 1]	(11) unlabeled('The ', [1, **1**, **3**, **2**, **1**], 6).
	(12) em(best, [2, **1**, **3**, **2**, **1**], 6).
	(13) unlabeled(' ever!', [3, **1**, **3**, **2**, **1**], 6).

The first index allows the recovering of the facts by means of the predicate name: `author`, `year`, and so on. Therefore the first index key is the name of the predicate and the first index value is the set of relative positions of the facts for the predicate. The second index allows to recover the relative position of the group in which a fact is included. Therefore the second index key is the number of the relative position of the fact and the second index value is the relative position of the group in which is included.

With this aim the first index stores for each predicate name annotations of the form $\texttt{pos}(\texttt{n},\texttt{m})$, in which `n` denotes the relative position of a fact for the predicate and `m` the relative position of the group of this fact.

For instance, `author` facts are stored in positions `1`, `2`, `3` and `9`, given by the annotation $\texttt{pos}(1,\_)$,$\texttt{pos}(2,\_)$,$\texttt{pos}(3,\_)$, $\texttt{pos}(9,\_)$, and the group of each author, that is, the record in which the author is included starts in positions `0`, `0`, `0` and `8`, respectively, given by the annotation $\texttt{pos}(1,\mathbf{0})$, $\texttt{pos}(2,\mathbf{0})$, $\texttt{pos}(3,\mathbf{0})$,$\texttt{pos}(9,\mathbf{0})$. Each "group" of facts has the same sequence of node numbers with the exception (but not always) of the first number. This common sequence of node numbers can be considered as the identifier of the group. For instance, w.r.t. the running example, the first group can be identified by $[1,1]$ and contains facts numbered with $[1,1]$, $[1,1,1]$, $[2,1,1]$, $[3,1,1]$ and $[4,1,1]$, the second of the group $[5,1,1]$, and so on. The reason for this grouping criteria is that each group of facts will be used in the same schema rule. For instance, in this case:

```
book(booktype(A, B, C, [D]), L ,2) :-
        author(A, [E|L],3),
        title(B, [F|L],3),
        review(C, [G|L],3),
        year(D, L,3).
```

and thus they are usually recovered at the same time. In Section 5, we will explain the combination of the evaluation method with the indexing.

4 Magic Set Transformation for XPath Expressions

In this section, we will present the magic set transformation technique for querying XPath expressions against an XML document represented by means of a logic program. With this aim, we will present the general criteria for the transformation technique.

4.1 Filtered rules

Since we use magic set transformations, firstly we need to add *magic filters* to each rule. For instance, with respect to our running example, we will consider the following set of rules:

Filtered Rules

```
books(bookstype(A, []),L,1) :-
        mg_books(bookstype(A, []),L,1),
        book(A, [C|L],2).
book(booktype(A, B, C, [D]),L,2) :-
        mg_book(booktype(A, B, C, [D]),L,2),
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(D,L,3).
review(reviewtype(A,B,[]),L,3):-
        mg_review(reviewtype(A,B,[]),L,3),
        unlabeled(A,[J|L],4),
        em(B,[K|L],4).
review(reviewtype(A,[]),L,3):-
        mg_review(reviewtype(A,[]),L,3),
        em(A,[K|L],5).
em(emtype(A,B,[]),L,5) :-
        mg_em(emtype(A,B,[]),L,5),
        unlabeled(A,[G|L],6),
        em(B, [H|L],6).
```

Filtered Facts

```
year('2003', [1, 1],3):- mg_year('2003', [1, 1],3).
author('Abiteboul', [1, 1, 1],3):-
        mg_author('Abiteboul', [1, 1, 1],3).
...
```

They are the so-called *filtered rules and facts* like typical magic-set based transformations.

4.2 Magic Transformation

Now, we present the general criteria for the specialization technique:

1. The handling of an XPath query involves the introduction of *one or more passing rules* and *one or more seeds*.

2. The seeds are generated from either the last element or the element situated before the first boolean condition on the XPath expression.

3. The passing rules are generated from the seed to terminal tags.

- For instance, we can assume a XPath query such as /books/book/author, requiring the authors in the book database. In this case, we have to generate the seed mg_author(X, Y, 3) where X, Y are logic variables, and the number 3 indicates the type number. The bottom-up evaluation of the filtered program from this seed will result on the following set of facts which represents the following XML document:

Computed Set of Facts	Represented XML Document
author('Abiteboul', [1, 1, 1],3). author('Buneman', [2, 1, 1],3). author('Suciu', [3, 1, 1],3). author('Buneman', [1, 2, 1],3).	<result> <author>Abiteboul</author> <author>Buneman</author> <author>Suciu</author> <author>Buneman</author> </result>

  In order to build the XML document, the idea is to consider the schema rules and the computed set of facts. In this case, author has no schema rules, and therefore the result can be built directly from the facts, considering the predicate name as tag and following the node numbering for the ordering of the XML elements.

- Now, we can assume the XPath expression /books/book. Now, the seed is mg_book(X, Y, 2), and since book has attributes and subelements, we need to generate the following *passing rules* which enable, from bottom-up evaluation, to generate, for each book, facts for author, title, review and year.

```
mg_author(A, [D|L] ,3) :-
        mg_book(booktype(A, E, F, [G]),L,2).
mg_review(A, [D|L],3) :-
        mg_book(booktype(E, F, A, [G]),L,2).
mg_title(A, [D|L],3) :-
        mg_book(booktype(E, A, F, [G]),L,2).
mg_year(A,L,3) :-
        mg_book(booktype(D, E, F, [A]),L,2).
mg_unlabeled(A,[J|L],4):-
        mg_review(reviewtype(A,B,[]),L,3).
mg_em(B,[K|L],4):-
        mg_review(reviewtype(A,B,[]),L,3).
mg_em(A,[K|L],5):-
        mg_review(reviewtype(A,[]),L,3).
mg_unlabeled(A,[G|L],6):-
        mg_em(emtype(A,B,[]),L,5).
mg_em(B, [H|L],6):-
        mg_em(emtype(A,B,[]),L,5).
```

They are built as usual in magic-set based transformations but in this case *without considering a left-to-right information passing strategy*. The bottom-up evaluation of the above filtered rules together these passing rules from the seed `mg_book(X,Y,Z,2)` is able to compute the following set of facts:

```
author('Abiteboul', [1, 1, 1], 3).
author('Buneman', [2, 1, 1], 3).
author('Suciu', [3, 1, 1], 3).
title('Data on the Web', [4, 1, 1], 3).
unlabeled('A', [1, 5, 1, 1], 4).
em('fine', [2, 5, 1, 1], 4).
unlabeled('book.', [3, 5, 1, 1], 4).
year('2003', [1, 1], 3).
author('Buneman', [1, 2, 1], 3).
title('XML in Scotland', [2, 2, 1], 3).
unlabeled('The', [1, 1, 3, 2, 1], 6).
em('best', [2, 1, 3, 2, 1], 6).
unlabeled('ever!', [3, 1, 3, 2, 1], 6).
year('2002', [2, 1], 3).
review(reviewtype('A','fine',[]),[5, 1, 1],3).
review(reviewtype('book','fine',[]),[5, 1, 1],3).
review(reviewtype(emtype('The','best',[]),[]),
[3, 2, 1],3).
review(reviewtype(emtype('ever!','best',[]),[]),
[3, 2, 1],3).
em(emtype('The','best',[]),[1, 3, 2, 1],5).
em(emtype('ever!','best',[]),[1, 3, 2, 1],5).
```

From this set of facts we can build the following answer in XML format:

```
<result>
<book year="2003">
<author>Abiteboul</author>
<author>Buneman</author>
<author>Suciu</author>
<title>Data on the Web</title>
<review>
A <em>fine</em> book.
</review>
</book>
<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
<review>
<em> The <em>best</em> ever!</em>
</review>
</book>
</result>
```

In order to build this XML document, we need to consider the following subset of schema rules, including the schema from book (i.e. the last element in the XPath expression) up to terminal tags:

```
book(booktype(A, B, C, [D]),L,2) :-
        author(A, [G|L],3),
        title(B, [H|L],3),
        review(C, [I|L],3),
        year(D, L,3).
review(reviewtype(A,B,[]),L,3):-
        unlabeled(A,[J|L],4),
        em(B,[K|L],4).
review(reviewtype(A,[]),L,3):-
        em(A,[K|L],5).
em(emtype(A,B,[]), L,5) :-
        unlabeled(A,[G|L],6),
        em(B, [H|L],6).
```

These schema rules together with the generated facts allow the reconstruction of the XML document representing the result. It is easy to build a program for writing the XML document representing the result into a file. The ideas is to follow the path from the last element to the XPath query up to terminal elements rebuilding the original structure, following the schema rules. Let us remark that the last group of facts, in bold style, are only computed as auxiliary results and they are not needed for the reconstruction of the XML document.

- With respect to the generated seeds, in general a set of seeds is generated, concretely one for each element to be retrieved. Next, we show different examples of XPath expressions with their corresponding generated seeds.

XPath Expression	Generated Seed
(1) /books/book/**author**	**(1)** mg_author(X, Y, 3)
(2) /books/**book**	**(2)** mg_book(X, Y, 2)
(3) /books/book/*	**(3)** mg_author(X, Y, 3) mg_title(X, Y, 3) mg_review(X, Y, 3)

In case (3), the XPath expression requests the subelements of `book`, and thus a seed for each one of them is generated.

- However whenever *a condition* is introduced, the seed is *forwarded* to the first occurrence of a condition. For instance, let us consider the XPath expression /`books`/`book`[`author` = `Suciu`]/`title`. In this case, we have a condition in the form of `author` = `Suciu`. Now, we have to generate the seed $\texttt{mg\_book}(\texttt{booktype}('\texttt{Suciu}', \texttt{A}, \texttt{B}), \texttt{C}, 2)$. That is, the seed is forwarded to the element situated before of the first boolean condition, in this case `book`. In addition, `mg_book` is instantiated with `Suciu` in the function symbol `booktype`, and, of course, in the position representing the authors. In addition, we have to consider the following passing rules:

```
mg_author(A,[D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2).
mg_title(A,[D|L],3) :-
        mg_book(booktype(E, A, F, [G]),L,2),
        author(E,[H|L],3).
```

In the bottom-up evaluation the seed will firstly trigger the retrieval of the author 'Suciu'. In particular, it retrieves the node numbers of Suciu's books. It is achieved due to the instantiation of the corresponding argument in the seed and the use of the first passing rule. Afterwards, it allows the retrieval of Suciu's book titles. With this aim, note that the predicate author(E, [H|L], 3) has been included *as condition for the passing rules* corresponding to the element title. It ensures that *Suciu's book titles are the only computed*. The use of author(E, [H|L], 3) is vital for efficient retrieval of such titles, given that the node number has been instantiated in this predicate in the first step and the information is passed to the predicate of title in the second step. It corresponds, in some sense, with a *left-to-right information passing strategy*. In this case, the generated fact is author('Suciu', [3, 1, 1], 3) in which the node number [3, 1, 1] is used for retrieving the fact title('Data on the Web', [4, 1, 1], 3). Next, we show the computed facts by means of the bottom-up evaluation as well as the XML document represented by these facts:

Computed Set of Facts	Represented XML Document
author('Suciu', [3,1,1],3). title('Data on the Web', [4,1,1],3).	<result> <title>Data on the Web</title> </result>

Let us remark that there is an additional computed fact: author('Suciu', [3, 1, 1], 3), which is not used for the building of the resulting XML document, but that it has been used for computing the relevant fact. The XML document can be directly built from the fact, given that title has no schema rules.

- In the case of *two or more conditions*, such as /books/book[@year = 2002 and title = Data on the Web]/author, we have to consider the seed mg_book (booktype(A,'Data on the Web', C, ['2002']), L, 2), as well as the following passing rules:

```
mg_year(A, L,3) :-
        mg_book(booktype(E, F, G, [A]), L,2).
mg_title(A, [D|L],3) :-
        mg_book(booktype(E, A, F, [G]),L,2),
        year(G,L,3).
mg_author(A, [D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2),
        year(G,L,3),
        title(G,[E|L],3).
```

In the passing rules, we follow a *left to right* strategy of the boolean condition. That is, starting from the seed mg_book(booktype(A,'Data on the Web',

C, ['2002']), L, 2), firstly the first passing rule triggers the retrieval of books for this year 2002. Afterwards, the second passing rule triggers the retrieval of titles of this year (using the node number instantiated in the previous step); concretely book titled "Data on the Web". Finally, the last passing rule retrieves the author of such books, using node numbers instantiated in the previous steps.

- In the case of a *boolean condition 'or'*, such as /books/book[@year = 2002 or title = Data on the Web]/author, we need to consider the following two seeds:

  (a) mg_book(booktype(A, B, C, ['2002']), L, 2) and

  (b) mg_book(booktype(A,'Data on the web', C, D), L, 2),

  together with the following passing rules:

```
mg_year(A, L,3) :-
        mg_book(booktype(E, F, G, [A]), L,2).
mg_title(A, [D|L],3) :-
        mg_book(booktype(E, A, F, [G]),L,2).
mg_author(A, [D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2),
        year(G,L,3).
mg_author(A, [D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2),
        title(G,[E|L],3).
```

  In this case, each seed triggers different searchings. The first seed together with the first passing rule triggers the retrieval of books of year 2002, and the second one together with the second passing rule, the retrieval of books titled "Data on the Web". Finally, authors are generated from both searchings.

- Whenever the occurrences of conditions are *at different level of the XPath query*, such as the XPath expression /books/book[@year = 2002]/author [name = Serge] with respect to the following XML document:

```
<books>
    <book year="2003">
        <author>Abiteboul<name>Serge</name></author>
        <title>Data on the Web</title>
        <review>A <em>fine</em> book.</review>
    </book>
    <book year="2002">
        <author>Buneman <name>Peter</name></author>
        <title>XML in Scotland</title>
    </book>
</books>
```

  then the seed is mg_book(booktype (authortype(A,'Serge', B), C, D, ['2002']), L, 2) and the passing rules are as follows:

```
mg_year(A,L,3) :-
        mg_book(booktype(E, F, G, [A]),L,2).
mg_author(A, [D|L],3) :-
        mg_book(booktype(A, E, F, [G]),L,2),
        year(G,L,3).
mg_name(A, [E|L],4) :-
        mg_author(authortype(A,G,[]),L,3).
```

Here, firstly, we filter the books by year, next we retrieve authors for these books, and finally author names are filtered and recovered.

XPath is a rich query language with many variants. The main cases have been shown. The handling of the rest of cases involves modifications on the number of seeds and the form of passing rules.

5 Combining Indexing and Magic Sets

Now, we would like to explain how the indexing technique presented in Section 3 is used for efficient recovering of the facts needed for answering a XPath query. For instance, let us assume the following query: /books/book[@year = 2002 and author = Buneman]/review w.r.t. the running example. Now, the transformation is as follows:

Filtered Rules

```
(1) books(bookstype(A,[]),B,1) :-
        mg_book(bookstype(A,[]),B,1),
        book(A,[C|B],2).
(2) book(booktype(A,B,C,[D]),[E,F],2) :-
        mg_book(booktype(A,B,C,[D]),E,2),
        author(A,[F|E],3),
        review(C,[G|E],3).
(3) review(reviewtype(A,B,[]),C,3):-
        mg_review(reviewtype(A,B,[]),C,3),
        unlabeled(A,[D|C],4),
        em(B,[E|C],4).
(4) review(reviewtype(A,[]),B,3):-
        mg_review(reviewtype(A,[]),B,3),
        em(A,[C|B],5).
(5) em(emtype(A,B,[]),C,5) :-
        mg_em(emtype(A,B,[]),C,5),
        unlabeled(A,[D|C],6),
        em(B, [E|C],6).
```

Seed

```
(6) mg_book(booktype('Buneman',A,B,['2002']),
        C,2).
```

Passing Rules

```
(7) mg_book(A,[B|C],2):-
        mg_books(bookstype(A,[]),C,1).
(8) mg_author(A,[B|C],3) :-
        mg_book(booktype(A, D, E, [F]),C,2),
        year(F,C,3).
(9) mg_year(A,B,3) :-
        mg_book(booktype(C,D,E,[A]),B,2).
(10) mg_review(A,[B|C],3) :-
        mg_book(booktype(D,E,A,[F]),C,2),
        year(F,C,3),
        author(D,[G|C],3).
(11) mg_unlabeled(A,[B|C],4):-
        mg_review(reviewtype(A,D,[]),C,3).
(12) mg_em(A,[B|C],4):-
        mg_review(reviewtype(D,A,[]),C,3).
(13) mg_em(A,[B|C],5):-
        mg_review(reviewtype(A,[]),C,3).
(14) mg_unlabeled(A,[B|C],6):-
        mg_em(emtype(A,D,[]),C,5).
(15) mg_em(A,[B|C],6):-
        mg_em(emtype(D,A,[]),C,5).
```

Filtered Facts

```
(16) year('2003',[1,1],3):-
        mg_year('2003',[1,1],3).
(17) author('Abiteboul',[1,1,1],3):-
        mg_author('Abiteboul',[1,1,1],3).
(18) author('Buneman',[2,1,1],3):-
        mg_author('Buneman',[2,1,1],3).
(19) author('Suciu',[3,1,1],3):-
        mg_author('Suciu',[3,1,1],3).
```

In the bottom-up evaluation, in general, the generated magic predicates have the form: $\texttt{mg\_tag}(\_, [\texttt{Var}_1, \ldots, \texttt{Var}_n, N_1, \ldots, N_m], \_)$, where tag is a label of the XML document, and the second argument is a list $[\texttt{Var}_1, \ldots, \texttt{Var}_n, N_1, \ldots, N_m]$

representing a partially instantiated node number, in which $\texttt{Var}_1, \ldots, \texttt{Var}_n$ are variables and $\texttt{N}_1, \ldots, \texttt{N}_m$ are natural numbers. There is a particular case: $\texttt{mg\_tag}(\_, \texttt{Var}, \_)$, in which there is a variable instead of a list. This particular case corresponds to the *seed*.

In addition, each time a fact is recovered, the system stores together the identifier of its group the relative position of the group (i.e. the starting point in the fact file of the group in which the element is stored). For instance, with regard to the example, if $\texttt{year}(2002, [2,1], 3)$ is recovered, the system stores that the group $[2,1]$ is stored from position 8.

Now, the index accessing can be summarized as follows: Each time a magic predicate $\texttt{mg\_tag}(\_, [\texttt{Var}_1, \ldots, \texttt{Var}_n, \texttt{N}_1, \ldots, \texttt{N}_m], \_)$ is generated then:

(a) Whenever $[\texttt{Var}_2, \ldots, \texttt{Var}_n, \texttt{N}_1, \ldots, \texttt{N}_m]$ matches to a previously stored relative position, the system uses the relative position of the group and the second index for the retrieval of facts for tag.

(b) Whenever the stored positions do not match to $[\texttt{Var}_2, \ldots, \texttt{Var}_n, \texttt{N}_1, \ldots, \texttt{N}_m]$, the system uses the first index for the retrieval of the elements of tag, and stores the new group position.

In the case $\texttt{tag}(\_, \texttt{Var}, \_)$ the first index will be used for recovering the facts.

Now, we show the trace of the execution of the above XPath query with respect to the indexing presented in Section 3.

- adding of $\texttt{mg\_book}(\texttt{booktype}(\texttt{Buneman}, \texttt{A}, \texttt{B}, [2002]), \texttt{C}, 2)$ (Rule 6)
- adding of $\texttt{mg\_year}(2002, \texttt{B}, 3)$ (Rule 9)
- first index accessing to position 0 due to $\texttt{mg\_year}(2002, \texttt{B}, 3)$; recovering $\texttt{year}(2003, [1,1], 3)$; storing that the group position of $[1,1]$ is 0.
- first index accessing to position 8 due to $\texttt{mg\_year}(2002, \texttt{B}, 3)$; recovering $\texttt{year}(2002, [2,1], 3)$; storing that the group position of $[2,1]$ is 8.
- adding of $\texttt{year}(2002, [2,1], 3)$ (Rule 24)
- adding of $\texttt{mg\_author}(\texttt{Buneman}, [\texttt{B}, 2, 1], 3)$ (Rule 8)
- second index accessing to position 8 due to $\texttt{mg\_author}(\texttt{Buneman}, [\texttt{B}, 2, 1], 3)$ and that the group position of $[2,1]$ is 8;
  recovering $\texttt{author}(\texttt{Buneman}, [1, 2, 1], 3)$
- adding of $\texttt{author}(\texttt{Buneman}, [1, 2, 1], 3)$ (Rule 25)
- adding of $\texttt{mg\_review}(\texttt{A}, [\texttt{B}, 2, 1], 3)$ (Rule 10)
- adding of $\texttt{mg\_unlabeled}(\texttt{A}, [\texttt{B}, \texttt{C}, 2, 1], 4)$ (Rule 11)

- adding of mg_em(A, [B, C, 2, 1], 4) (Rule 12)
- adding of mg_em(A, [B, C, 2, 1], 5) (Rule 13)
- adding of mg_unlabeled(A, [B, C, D, 2, 1], 6) (Rule 14)
- first index accessing to position 11 due to mg_unlabeled(A, [B, C, D, 2, 1], 6); recovering unlabeled (The, [1, 1, 3, 2, 1], 6); storing that the group position of [1, 3, 2, 1] is 11.
- adding of unlabeled(The, [1, 1, 3, 2, 1], 6) (Rule 27)
- first index accessing to position 13 due to mg_unlabeled(A, [B, C, D, 2, 1], 6); recovering unlabeled (ever!, [3, 1, 3, 2, 1], 6); storing that the group position of [1, 3, 2, 1] is 11.
- adding of unlabeled(ever!, [3, 1, 3, 2, 1], 6) (Rule 29)
- adding of mg_em(A, [B, C, D, 2, 1], 6) (Rule 15)
- second index accessing to position 11 due to mg_em(A, [B, C, D, 2, 1], 6) and that the group position of [1, 3, 2, 1] is 11; recovering em(best, [2, 1, 3, 2, 1], 6)
- adding of em(best, [2, 1, 3, 2, 1], 6) (Rule 28)
- adding of em(emtype(The, best, [ ]), [1, 3, 2, 1], 5) (Rule 5)
- adding of em(emtype(ever!, best, [ ]), [1, 3, 2, 1], 5) (Rule 5)
- adding of review(reviewtype(emtype(The, best, [ ]), [ ]), [3, 2, 1], 3) (Rule 4)
- adding of review(reviewtype(emtype(ever!, best, [ ]), [ ]), [3, 2, 1], 3) (Rule 4)
- adding of book(booktype(Buneman, A, reviewtype(emtype(The, best, [ ]), [ ]), [2002]), [2, 1], 2) (Rule 2)
- adding of book(booktype(Buneman, A, reviewtype(emtype(ever!, best, [ ]), [ ]), [2002]), [2, 1], 2) (Rule 2)

6 XINDALOG Prototype

In this section, we briefly show our prototype, named XINDALOG. This prototype has been developed under SWI-Prolog [Wie05] and hosted in a web site at http://indalog.ual.es/Xindalog. This web site has been developed by using a CGI (Common Gateway Interface) application in order to link the web site with the prototype. From the main page of the web site, you can access to a

basic description of XINDALOG, `XML`, `XPath`, as well as the `demo`. We have implemented two releases of the prototype: a `bottom-up` and a `top-down` version. In the web site, there are some examples to be tested.

Figure 2: `http://indalog.ual.es/Xindalog`

Figure 3: Bottom-up demo

Figure 4: Query example

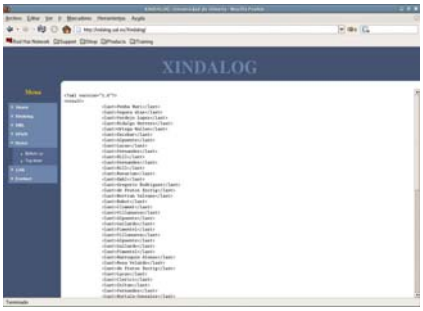

Figure 5: Query result

7 Examples of XPath Queries in XINDALOG

Our prototype has been tested by means of complex XPath queries and not too structured XML documents. The following table shows XPath examples tested w.r.t. the following document.

XPath Query	Meaning
⊙ /books/book[@year and @pages]/*	To obtain the books which have publishing year and number of pages
⊙ /books/book/author/@*	To obtain all the attributes of the authors
⊙ //book	To obtain all the books included in the XML document
⊙ //book[review="Very good"]/author	To obtain all the authors of books with a very good review
⊙ //@year	To obtain all the years occurring in the XML document
⊙ /books/*/author	To obtain all the authors inside book records
⊙ /books/book[review="Good"]/ author[name="John Durant"]	To obtain all the author information whose name is John Durant and the review is good

```
<books year="2006">
    A book collection
    <book>empty</book>
    <book year="2003" pages="984">
        The first book
        <author english="yes" spanish="yes">
            Benz
            <name>Brian</name>
        </author>
        <author>John Durant</author>
        <author>John Durant</author>
        <title>XML Programming Bible</title>
        <review>Good</review>
    </book>
    <book year="2002">
        The second book
        <author>Dino Esposito</author>
        <title>Applied XML Programming for Microsoft .NET</title>
        <review>Good</review>
    </book>
    <book>
    The third book
        <author>Apt, Krzystof R.</author>
        <title>The Logic Programming Paradigm and Prolog</title>
        <review>Very good</review>
    </book>
    <book year="1994" pages="560">
        The fourth book
        <author english="yes" spanish="no">
            Leon Sterling
        </author>
        <author>Ehud Shapiro</author>
        <title>The Art of Prolog</title>
        <review>Very good</review>
    </book>
    <book2 year="2001">
        The fifth book
        <author english="yes">
            Elliotte Rusty Harold
        </author>
        <title>XML Bible</title>
        <review2>Good</review2>
    </book2>
    <book year="2003" pages="984">
        The first book
        <author english="yes" spanish="yes">
            Benz
            <name2>Brian</name2>
            <firstname>
            Brian
            <lastname>Benz</lastname>
            <others>no more</others>
            </firstname>
        </author>
        <author>John Durant</author>
        <author>John Durant</author>
        <title>XML Programming Bible</title>
        <review>Very good 2</review>
    </book>
</books>
```

Table 1: A small XML document

XPath Query	Meaning
⊙ /books[book="The first book"]/book [@year="2003" and review="Good"] /author[name="Benz"]/../..	To obtain the books of the year 2003 and good review whose author is Benz
⊙ /books/book/text()	To obtain the books with textual information
⊙ /books/book[author/name]/title	To obtain the book titles whenever the books have author name

XPath Query	Meaning
⊙ /books/(book | book2)/(review2 | review)	To obtain the reviews of the two kinds of books
⊙ /books/book/(author | title)	To obtain the book authors and titles
⊙ /books/(book | book2)//text()	To obtain the textual information from the two kinds of books
⊙ //@*	To obtain all the attributes of the document
⊙ /*/*/title	To obtain the titles that are at 3rd level
⊙ /*/*//*	To obtain all the elements and their nested from the 3rd level
⊙ /*/book2/*	To obtain all information from book2 at 2nd level
⊙ //*//author/..	To obtain the records containing author information from the 1st level

8 Benchmarks of XPath Queries in XINDALOG

In this section, we would like to show benchmarks of XPath queries under the XINDALOG prototype.

Our prototype has been tested by means of XML documents of big size. This test allows us to know if we get reasonable benchmarks when extensive disk accesses are needed. We have considered the following file sizes: 64KB, 128KB, 256KB, 512KB, and, finally, 1024KB. For each file size, we have computed the following answer times:

- Translation time; it represents the time needed for translating a `XML` document into `PROLOG` facts and rules;
- Evaluation time; it represents the time of the bottom-up evaluation of the transformed program w.r.t. a XPath query;
- Visualization time; it represents the time needed for formatting and browsing the query result.

We have considered the XPath query `/books/book[review="good"]/title`, which requests only the titles of those books whose review is `good`. For this query, we consider the following cases:

- *The program is not specialized*, that is, we generate all the passing rules without taking into account the XPath expression. However, the seed is instantiated w.r.t. the XPath query.
- *The program is specialized*, that is, we generate only the passing rules needed for the XPath query. The seed is also instantiated.

With Program Specialization

XPath Query: /books/book[review="good"]/title

File size	Translation	Evaluation	Browsing	Total time
64KB	0,750sg	0,973sg	0,017sg	1,704sg
128KB	2,171sg	3,344sg	0,031sg	5,546sg
256KB	6,485sg	12,467sg	0,063sg	19,015sg
512KB	22,312sg	47,859sg	0,125sg	1min 10,296sg
1024KB	1min 21,563sg	3min 9,453sg	0,234sg	4min 31,250sg

Without Program Specialization

XPath Query: /books/book[review="good"]/title

File size	Translation	Evaluation	Browsing	Total time
64KB	0,780sg	7,563sg	0,079sg	8,422sg
128KB	2,095sg	28,843sg	0,157sg	31,095sg
256KB	6,579sg	1min 51,108sg	0,297sg	1min 57,984sg
512KB	22,468sg	7min 15,142sg	0,500sg	7min 38,110sg
1024KB	1min 21,735sg	28min 49,906sg	1,219sg	30min 12,860sg

Once shown the answer times, we think that there exists an important improvement of answer times by considering the program specialization w.r.t. the proposed query. Obviously, as was commented previously, our specialization program technique improves the answer times whenever only certain parts of a XML document are demanded. In fact, if we consider the XPath query /books, wherein the full document is demanded, then the program specialization has no effect, and thus the answer times with and without program specialization are the same. In this case, the answer times are the following:

With and Without Program Specialization

XPath Query: /books

File size	Translation	Evaluation	Browsing	Total time
64KB	0,750sg	1,500sg	0,062sg	2,312sg
128KB	2,110sg	5,485sg	0,140sg	7,735sg
256KB	6,562sg	21,047sg	0,235sg	27,844sg
512KB	22,328sg	1min 23,470sg	0,422sg	1min 46,220sg
1024KB	1min 24,407sg	5min 29,203sg	0,843sg	6min 54,453sg

Other query examples with their corresponding answer times requiring program specialization w.r.t. the query are the following:

XPath Query: `/books/book[review="very good"]/title`

File size	Translation	Evaluation	Browsing	Total time
64KB	0,750sg	0,031sg	0,0sg	0,781sg
128KB	2,094sg	0,031sg	0,016sg	2,141sg
256KB	6,486sg	0,062sg	0,0sg	6,548sg
512KB	22,640sg	0,110sg	0,0g	22,750sg
1024KB	1min 24,937sg	0,173sg	0,015sg	1min 25,125sg

Note that the XPath query (a) `/books/book[review="very good"]/title` is very similar to the XPath query (b) `/books/book[review="good"]/title`. However, there exists an important difference in the evaluation time for all file sizes. What is the reason of this difference? The reason is due to the number of elements to be retrieved. In the case (a), there exist a lot of elements with `review = good` in the document and thus the query needs to retrieve a big number of elements. Whereas, the number of elements with `review = very good` is too smaller, and thus, in the case (b), we need to retrieve a smaller number of elements.

XPath Query: `/books/book/title`

File size	Translation	Evaluation	Browsing	Total time
64KB	0,766sg	0,063sg	0,016sg	0,845sg
128KB	2,077sg	0,158sg	0,015sg	2,250sg
256KB	6,625sg	0,453sg	0,032sg	7,110sg
512KB	22,345sg	1,453sg	0,047g	23,845sg
1024KB	1min 22,203sg	5,468sg	0,079sg	1min 27,750sg

Finally, in this query, we are demanded all elements `title` inside elements `book` occurring in the `XML` document. In this case, in spite of requiring a big number of elements, `the evaluation time` is reasonably good. Remark that this query has no conditions, and thus no extra evaluations for checking a possible condition are needed.

9 Conclusions and Future Work

In this paper, we have presented how to represent and index XML documents by means of logic programming. Moreover, we have studied how to transform such programs by means of the magic set transformations and how to combine the bottom-up evaluation with the indexing technique in order to obtain the answers w.r.t. an XPath query. As future work, we will study how to extends our work in order to translate a XQuery query into logic programming.

Acknowledgements

We would like to thank anonymous referees for their useful comments. In addition, we would like to remark that this work has been partially supported by the Spanish MCYT under grant TIC2002-03968, the Spanish MEC under grant TIN2005-09207-C03-02, and EU (FEDER).

References

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. M. Kaufmann, 2000.

[ABS01] J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Funtional Logic Deductive Databases. In *Proc. of ICLP*, LNCS 2237, pages 331–347. Springer, 2001.

[Apt90] K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, chapter 10, pages 493–574. MIT Press, 1990.

[BFG01] R. Baumgartner, S. Flesca, and G. Gottlob. The Elog Web Extraction Language. In *Proc. of International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, LNCS 2250, pages 548–560. Springer, 2001.

[Bol01] H. Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Proc. of INAP*, pages 124–139. Prolog Association of Japan, 2001.

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *JLP*, 10(3,4):255–299, 1991.

[BS02] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. of ICLP*, LNCS 2401, pages 255–270. Springer, 2002.

[CF03] J. Coelho and M. Florido. Type-based xml processing in logic programming. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, pages 273–285. LNCS 2562, 2003.

[CH01] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *TPLP*, 1(3):251–282, 2001.

[Cha02] D. Chanberlin. XQuery: An XML Query Language. *IBM Systems Journal*, 41(4):597–615, 2002.

[HP03] H. Hosoya and B. C. Pierce. XDuce: A Statically Typed XML Processing Language. *TOIT*, 3(2):117–148, 2003.

[May04] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *TPLP*, 4(3):239–287, 2004.

[MS03] A. Marian and J. Simeon. Projecting XML Documents. In *Proc. of VLDB*, pages 213–224. Morgan Kaufmann, 2003.

[SB02] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of RuleML*, 2002.

[SW03] J. Simeon and P. Wadler. The Essence of XML. In *Proc. of POPL*, volume 38, pages 1–13. ACM, 2003.

[W3C01] W3C. XML Schema 1.0. Technical report, www.w3.org, 2001.

[W3C04a] W3C. OWL Ontology Web Language. Technical report, www.w3.org, 2004.

[W3C04b] W3C. Resource Description Framework (RDF). Technical report, www.w3.org, 2004.

[W3C04c] W3C. XML Path Language (XPath) 2.0, Draft. Technical report, www.w3.org, 2004.

[W3C04d] W3C. XQuery 1.0: An XML Query Language. Technical report, www.w3.org, 2004.

[W3C04e] W3C. XSL Transformations (XSLT) Version 2.0. Technical report, www.w3.org, 2004.

[Wad02] P. Wadler. XQuery: A Typed Functional Language for Querying XML. In *AFP*, LNCS 2638, pages 188–212. Springer, 2002.

[Wie05] J. Wielemaker. SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam, March 2005.