Fuzzy XPath Queries in XQuery

Jesús M. Almendros-Jiménez[1], Alejandro Luna[2], and Ginés Moreno[2]

[1] Dep. of Informatics, University of Almería, Spain
Email: jalmen@ual.es
[2] Dep. of Computing Systems, University of Castilla-La Mancha, Spain
Emails: Gines.Moreno@uclm.es, Alejandro.Luna@alu.uclm.es

Abstract. We have recently designed a fuzzy extension of the XPath language which provides ranked answers to flexible queries taking profit of fuzzy variants of *and*, *or* and *avg* operators for XPath conditions, as well as two structural constraints, called *down* and *deep*, for which a certain degree of relevance is associated. In this work, we describe how to implement the proposed fuzzy XPath with the XQuery language. Basically, we have defined an XQuery library able to fuzzily handle XPath expressions in such a way that our proposed fuzzy XPath can be encoded as XQuery expressions. The advantages of our approach is that any XQuery processor can handle a fuzzy version of XPath by using the library we have implemented.

Keywords: Fuzzy XPath; XML; XQuery

1 Introduction

The *XPath* language [6] has been proposed as standard for XML querying and it is based on the description of the path in the XML tree to be retrieved. XPath allows to specify the name of nodes (i.e., tags) and attributes to be present in the XML tree together with Boolean conditions about the content of nodes and attributes. XPath querying mechanism is based on a Boolean logic: the nodes retrieved from an XPath expression are those matching the path of the XML tree, according to Boolean conditions.

Information retrieval requires the design of query languages able to adapt to user's preferences and providing ranked sets of answers. The *degree of satisfaction* of the user with respect to an answer can be measured in several ways. XPath lacks on mechanisms for giving priority to queries and ranking answers. In an XPath-based query, the main criteria to provide a certain degree of satisfaction are the *hierarchical deepness* and *document order*. Moreover, conditions on XPath expressions are usually of varying importance for a user, that is, the user gives a *higher degree of importance to certain requirements* when satisfying his (her) wishes.

With this aim we have recently designed a fuzzy extension of XPath whose main aim is to provide mechanisms to assign priority to queries and to rank answers. Priorities are given by using fuzzy extensions of Boolean operators, while rankings are defined with regard to the location of a tag in the XML tree.

Firstly, we have proposed the incorporation to XPath of two structural constraints called *down* and *deep* for which a certain degree of relevance can be associated. So, whereas *down* provides a ranked set of answers depending on the path they are found

from "top to down" in the XML document, *deep* provides a ranked set of answers depending on the path they are found from "left to right" in the XML document. Both structural constraints can be used together, assigning degree of importance with respect to the distance to the root XML element.

Secondly, we provide fuzzy variants of *and* and *or* for XPath conditions. We have enriched the arsenal of operators of XPath with fuzzy variants of *and* and *or*. Particularly, we have considered three versions of *and*: *and+*, *and*, *and-* (and the same for *or* : *or+*, *or*, *or-*) which make more flexible the composition of fuzzy conditions. Three versions for each operator which are obtained from our adaptation of the *Product*, *Łukasiewicz* and *Gödel* logics to the XPath paradigm. We claim that in our work the fuzzy versions provide a mechanism to force (and debilitate) conditions in the sense that stronger (and weaker) user preferences can be modeled with the use of stronger (and weaker) fuzzy conditions. The combination of fuzzy operators in queries permits to specify a ranked set of fuzzy conditions according to user's requirements.

Finally, we have equipped XPath with an additional operator that is also traditional in fuzzy logic: the average operator *avg*. This operator offers the possibility to explicitly give weight to fuzzy conditions. Rating conditions by *avg*, solutions increase their weight in a proportional way. However, from the point of view of the user's preferences, it forces the user to quantify his (her) wishes which, in some occasions, can be difficult to measure. For this reason, fuzzy versions of *and* and *or* are better choices in some circumstances.

In this work, we describe how to implement our fuzzy variant of the XPath language within the XQuery language. Basically, we have defined an XQuery library able to fuzzily handle XPath expressions in such a way that our proposed fuzzy XPath can be encoded as XQuery expressions.

The implementation of our fuzzy extension of XPath is based on an XQuery library of functions including *deep* and *down* operators, as well as the fuzzy operators *and+*, *and-*, *and*, *or+*, *or-*, *or* and *avg*. Using this library the user can replace Boolean operators by fuzzy versions in XPath expressions, as well as he (she) can call to *deep* and *down* operators, in order to obtain ranked sets of answers. The answers are shown with a *Retrieval State Value (RSV)* representing the degree of satisfaction. The answers can also be ordered with respect to the RSV making use of *descending* XQuery expression, as well as filtered with regard to a *threshold*.

The input documents in our proposal are *crisp XML documents*, but the answers to a query offer fuzzy information, that is, a *RSV for each answer*. Therefore our approach is focused on the handling of standard XML documents, in which the user can retrieve information, ranked by a certain degree of satisfaction. We have decided to implement fuzzy XPath within XQuery by providing an XQuery library of fuzzy operators. It makes possible that our library can be used from any XQuery processor to query any XML document with crisp information.

Although the input of a query is a crisp XML document, the library assign internally and, in a transparent way to the user, a RSV to each of node of interest in the document. The RSVs assigned to each node of interest are used to compute the RSV of the answer. It makes the implementation a non-trivial task. Starting from a crisp XML document as input, our implementation annotates at run-time a RSV to each node of the query result. It also involves to dynamically annotate RSVs of nodes in subqueries. Additionally, *where* and *return* expressions of XQuery become XQuery functions in order to handle fuzzy conditions and RSVs, respectively.

Fig. 1. Fuzzy XPath Grammar

xpath :=	['['deep-down']']path
path :=	literal | text() | node | @att | node/path | node//path
node :=	QName | QName[cond]
cond :=	xpath op xpath | xpath num-op number
deep :=	**DEEP**=number
down :=	**DOWN**=number
deep-down :=	deep | down | deep '**;**' down
num-op :=	$> \mid = \mid < \mid <>$
fuzzy-op :=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{number,number}
op :=	num-op | fuzzy-op

Finally, let us remark that we have previously developed in [1–5] an implementation[3] of our fuzzy XPath using the *FLOPER "Fuzzy LOgic Programming Environment for Research"* tool[4] which is based on Multi-Adjoint Logic Programming (MALP) [21, 22]. There we made use of the fuzzy logic nature of FLOPER to implement fuzzy XPath by using fuzzy logic rules. Here the implementation has to adapt a Boolean logic based language (i.e., XQuery) to obtain the same behavior as in MALP. The implementation in XQuery can be downloaded from `http://dectau.uclm.es/fuzzyXPath/`.

The structure of the paper is as follows. Section 2 will describe the fuzzy version of XPath. Section 3 will show some examples of fuzzy XPath. Section 4 will present the implementation in XQuery. Section 5 will show the same of examples than Section 3, written in XQuery. Section 6 will present related work. Finally, Section 7 will conclude and present future work.

2 A Flexible XPath Language

Our proposal of fuzzy XPath is defined by the grammar of Figure 1. Basically, the extension of XPath is as follows:

- A given XPath expression can be adorned with «[DEEP = r_1; DOWN = r_2]» which means that the *deepness* of elements is penalized by r_1 and that the *order* of elements is penalized by r_2, and such penalization is proportional to the distance (i.e., the length of the branch and the weight of the tree, respectively). In particular, «[DEEP = 1; DOWN = r_2]» can be used for penalizing only w.r.t. document order. *deep* works for //, that is, the deepness in the XML tree is only computed when descendant nodes are explored, while *down* works for both / and //. Let us remark that *deep* and *down* can be used anywhere, and many times in an XPath expression.
- We consider three versions for each one of the conjunction and disjunction operators (also called connectives or aggregators) which are based in the so-called *Product*, *Gödel* and *Łukasiewicz* fuzzy logics. The *Gödel* and *Łukasiewicz* logic based fuzzy symbols [5] are represented in our application by *and+*, *and-*, *or-* and *or+*,

[3] `http://dectau.uclm.es/fuzzyXPath/`

[4] `http://dectau.uclm.es/floper`.

[5] The fuzzy logic community frequently uses the terms *t-norm* and *t-conorm* for expressing generalized versions of conjunctions and disjunctions.

Fig. 2. Fuzzy Logical Operators

$\&_{\mathtt{P}}(x,y) = x * y$	$\|_{\mathtt{P}}(x,y) = x + y - x * y$	*Product: and/or*
$\&_{\mathtt{G}}(x,y) = \min(x,y)$	$\|_{\mathtt{G}}(x,y) = \max(x,y)$	*Gödel: and+/or-*
$\&_{\mathtt{L}}(x,y) = \max(x+y-1,0)$	$\|_{\mathtt{L}}(x,y) = \min(x+y,1)$	*Łukasiewicz: and-/or+*

in contrast with product logic operators *and* and *or* (see Figure 2). Adjectives like *pessimistic*, *realistic* and *optimistic* are sometimes applied to the *Łukasiewicz*, *Product* and *Gödel* fuzzy logics since operators satisfy that, for any pair of real numbers x and y in $[0,1]$: $0 \leq \&_{\mathtt{L}}(x,y) \leq \&_{\mathtt{P}}(x,y) \leq \&_{\mathtt{G}}(x,y) \leq 1$ and the contrary for the disjunction operations: $0 \leq |_{\mathtt{G}}(x,y) \leq |_{\mathtt{P}}(x,y) \leq |_{\mathtt{L}}(x,y) \leq 1$. So, note that it is more difficult to satisfy a condition based on a pessimistic conjunctor/disjunctor (i.e, *and-/or-* inspired by the *Łukasiewicz* and *Gödel* logics, respectively) than with *Product* logic based operators (i.e, *and/or*), while the optimistic versions of such connectives (i.e., *and+/or+*) are less restrictive, obtaining a greater set of answers. This is a consequence of the following chain of inequalities: $0 \leq and-(x,y) \leq and(x,y) \leq and+(x,y) \leq or-(x,y) \leq or(x,y) \leq or+(x,y) \leq 1$. Therefore users should refine queries by choosing operators in the previous sequence from left to right (or from right to left), till finding solutions satisfying in a stronger (or weaker) way the requirements.

- Finally, the *avg* operator is defined too in a *weighted* way. Assuming two given *RSV*'s r_1 and r_2, *avg* is defined as $(r_1+r_2)/2$, and $avg\{a,b\}$ is defined as $(a*r_1+b*r_2)/a+b$.

In general, an extended XPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated RSV. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new RSV from the RSVs of the involved XPath expressions, which at the same time, provides a RSV to the node.

2.1 Examples of Fuzzy XPath

In order to illustrate the language, let us see some examples of flexible queries in XPath. We will take as input document the one shown in Figure 3. The example shows a sequence of hotels where each one is described by *name* and *price*, proximity to streets (*close_ to*[6]) and provided services (*pool* and *metro* -together with distance-). In the example, we assume that *document order* has the following semantics[7]. The tag *close_ to* specifies the proximity to a given street. However, the order of *close_ to* tags is relevant, and the top streets are closer than the streets at the bottom. In other words, the case:

```
hotel_H
        close_to street_A
        close_to street_B
```

[6] Let us remark that *close_ to* is not a fuzzy relation in our approach. As was commented before, input XML documents are crisp.

[7] The document order semantics can vary from one document to another. This example has been chosen to show the expressive power of our query language. Another kind of document can have a different order semantics and therefore the queries should be adapted.

Fig. 3. Input XML document collecting Hotel's information

```
<hotels>
<hotel name="Melia">
    <close_to>Gran Via
        <close_to>Callao</close_to>
        <close_to>Plaza de Espana</close_to>
    </close_to>
    <services>
        <pool></pool>
        <metro>150</metro>
    </services>
    <price>100</price>
</hotel>
<hotel name="NH">
    <close_to>Sol
        <close_to>Gran Via</close_to>
        <close_to>Callao</close_to>
    </close_to>
    <services>
        <metro>300</metro>
    </services>
    <price>150</price>
</hotel>
<hotel name="Hilton">
    <close_to>Moncloa
        <close_to>Gran Via</close_to>
        <close_to>Sol</close_to>
    </close_to>
    <services>
        <metro>150</metro>
    </services>
    <price>50</price>
</hotel>
<hotel name="Tryp">
    <close_to>Cibeles
        <close_to>Alcala
            <close_to>Gran Via</close_to>
        </close_to>
        <close_to>Retiro</close_to>
    </close_to>
    <services>
        <pool></pool>
        <metro>10</metro>
    </services>
    <price>575</price>
</hotel>
<hotel name="Sheraton">
    <close_to>Recoletos
        <close_to>Cibeles</close_to>
        <close_to>Gran Via
            <close_to>Sol</close_to>
        </close_to>
    </close_to>
    <close_to>Sol</close_to>
    <services>
        <pool></pool>
        <metro>300</metro>
    </services>
    <price>475</price>
</hotel>
</hotels>
```

implies that hotel H is near to both streets A and B, but closer to A than to B. The nesting of *close_to* has also a relevant meaning. While a given street A can be close to the hotel H, the streets close to A are not necessarily close to the hotel H. In other words, in the case:

```
        hotel_H
                close_to street_A
                                close_to street_B
```

the street B is near to street A, and street A is close to hotel H, which implies that street B is also close to hotel H, but no so close as street A. For instance, H can be situated at the end of street A, and B can cross A at the beginning. We can say, in this case, that B is an *adjacent* street to H, while A is *close* to H. This means that when looking for a hotel close to a given street, the highest priority should be assigned to streets close to the hotel, while adjacent streets should be relegated to lower priority.

Particularly, when the user tries to find hotels very close to a given street a high *down* value and a low *deep* value should be provided, whereas in the case the user tries to find hotels in the neighborhood of an street, a high *deep* and low *down* should be requested.

In our first example, we focus on the use of *down*. Let us now suppose that the user is interested to find a hotel close to Sol street. This might be his (her) first tentative looking for a hotel. Using crisp XPath he (she) would formulate:

$$<< /hotels/hotel[close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

However, it gives the user the set of hotels close to Sol without distinguishing the degree of proximity. The fuzzy version of XPath permits to specify a degradation of answers, in such a way that the user reformulates the query as:

$$<< /hotels/hotel[[DOWN = 0.9]close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

The query specifies that *close_ to* tag is degraded by 0.9 from top to down. In other words, when Sol is found close to a hotel, the position in which it occurs gives a different satisfaction value. In this case, we will obtain:

```
<result>
  <result rsv="1.0">NH</result>
  <result rsv="0.9">Sheraton</result>
</result>
```

Fortunately, we have found a hotel (NH) which is very close to Sol, and one (Sheraton) which is a little bit farther from Sol. Let us remark the previous example and the other examples of the Section show the results in order of satisfaction degree.

Let us now suppose that we are looking for a hotel close to Callao. In this case, we can try to make the same question:

$$<< /hotels/hotel[[DOWN = 0.9]close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

However, the result is empty. Therefore we can try to relax the query by changing '/' by '//':

$$<< /hotels/hotel[[DOWN = 0.9]//close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

Now, we will find answers, however, we will not be able to distinguish the proximity of the hotels. Our fuzzy version of XPath permits to specify how the solutions are degraded but not only taking into account the order but also the deepness. In other words, there would be useful to give different weights to be a close street, and to be an adjacent street. Therefore we can use the query:

$$<< /hotels/hotel[[DEEP = 0.5; DOWN = 0.9]//close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

obtaining the following results:

```
<result>
  <result rsv="0.5">Melia</result>
  <result rsv="0.45">NH</result>
</result>
```

Thus Melia is near to Callao, and NH is a little bit farther than Melia.

The use of *deep* combined with *down* could be considered as the best choice. However, *deep* can be used alone when the user only wants to penalize adjacency. Whenever we like to search hotels near to Gran Via street, degrading adjacent streets with a factor of 0.5, we can consider the following query (and we obtain the following result):

$$<< //hotel[[DEEP = 0.5]//close\_to/text() = \text{“}GranVia\text{”}]/@name >>$$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.5">NH</result>
  <result rsv="0.5">Hilton</result>
  <result rsv="0.5">Sheraton</result>
```

```
  <result rsv="0.25">Tryp</result>
</result>
```

We can see that *Melia* is close to *Gran Via*, while *NH*, *Hilton* and *Sheraton* are situated in adjacent streets of *Gran Via*. *Tryp* is the farthest hotel.

Let us now suppose that the user is interested in a hotel combining two services like pool and metro. Instead of using classical *and/or* connectives for mixing both features, we can obtain more flexible estimations on *RSV* values by using the *avg* operator as follows:

$$<< //hotel[services/pool\ avg\ services/metro]/@name >>$$

thus obtaining the following results:

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="1.0">Tryp</result>
  <result rsv="1.0">Sheraton</result>
  <result rsv="0.5">NH</result>
  <result rsv="0.5">Hilton</result>
</result>
```

By using the *avg* fuzzy operator, the user finds that *Melia*, *Tryp* and *Sheraton* have pool and metro, while *NH* and *Hilton* lack on one of them.

Let us now suppose that the importance of the metro is the double of the importance of the pool. In this case, the user can formulate the query as follows:

$$<< //hotel[services/pool\ avg\{1,2\}\ services/metro]/@name >>$$

obtaining the following results:

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="1.0">Tryp</result>
  <result rsv="1.0">Sheraton</result>
  <result rsv="0.666667">NH</result>
  <result rsv="0.666667">Hilton</result>
</result>
```

We can see in the results that NH and Hilton increase the degree of satisfaction w.r.t. the previous query given that they have metro station.

Let us now suppose the user is looking now for hotels giving more importance to the fact that the price of the hotel is lower than 150 euros than to the proximity to Sol street. The user can formulate the query as follows, obtaining the results below:

$$<< //hotel[[DEEP = 0.8]//close\_to/text() = \text{“}Sol\text{”}\ avg\{1,2\}\ //price/text() < 150]/@name >>$$

```
<result>
  <result rsv="0.933333">Hilton</result>
  <result rsv="0.666667">Melia</result>
  <result rsv="0.333333">NH</result>
  <result rsv="0.333333">Sheraton</result>
</result>
```

In the following queries we express the following requirement: hotels near to Gran Via, near to a metro station, having pool, with greater preference (3 to 2) to pool than metro. We will use *and+*, *and* and *and-* which provide different levels of exigency, which are demonstrated in the results.

$$<< //hotel[([DEEP = 0.5]//close\_to/text() = \text{”}GranVia\text{”})\ and+(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.5">Sheraton</result>
  <result rsv="0.4">Hilton</result>
```

```
  <result rsv="0.25">Tryp</result>
</result>
```

$<< //hotel[([DEEP = 0.5]//close\_to/text() = "GranVia")\ and(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.3">Sheraton</result>
  <result rsv="0.25">Tryp</result>
  <result rsv="0.2">Hilton</result>
</result>
```

$<< //hotel[([DEEP = 0.5]//close\_to/text() = "GranVia")\ and - (//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.25">Tryp</result>
  <result rsv="0.1">Sheraton</result>
</result>
```

So, in the first case (the least demanding and optimistic) we obtain four hotels (Melia, Sheraton, Hilton and Tryp), as well as in the second case (a little bit more exigent) while third table (the strongest one) lists three candidates (Melia, Tryp and Sheraton). Sheraton and Hilton are degraded using *and* and *and-*.

3 XQuery Library for Fuzzy XPath

We can summarize the elements of the implementation as follows:

3.1 Elements of the Library

1. The *deep* and *down* operators become XQuery functions that take as arguments a context node, an XPath expression and the value (a real number in [0,1]) assigned to deep and down, respectively. For combining deep and down an XQuery function is defined having as argument two real values in [0,1]:

```
declare function f:deep($node,$xpath,$deep)
declare function f:down($node,$xpath,$down)
declare function f:deep_down($node,$xpath,$deep,$down)
```

2. Fuzzy versions of Boolean operators *and*, *or* have been defined as XQuery functions, each one for each fuzzy logic we have considered (i.e., *Product*, *Łukasiewicz* and *Gödel*):

```
declare function f:andP($left,$right)
declare function f:orP($left,$right)
declare function f:andG($left,$right)
declare function f:orG($left,$right)
declare function f:andL($left,$right)
declare function f:orL($left,$right)
```

3. Operators *avg* and *avg{a,b}* have been defined as XQuery functions:

```
declare function f:avg($left,$right)
declare function f:avg_ab($left,$right,$a,$b)
```

4. Fuzzy versions of XQuery expressions *where* and *return* have been defined. In order to make transparent to the user the incorporation of RSVs, we have defined a new version of the *return* expression, called *returnF*, which transparently carries out the computation of the RSVs of the answers. Similarly, since XQuery works with a Boolean logic, the introduction of fuzzy versions of the operators, force us to define a new version of the *where* expression, called *whereF*, which transparently carries out the computation of the RSVs from fuzzy conditions. *ReturnF* has as parameters the context node and an XPath expression. *WhereF* has as parameters the context node and a fuzzy condition.

```
declare function f:whereF($node,$fuzzycond)
declare function f:returnF($node,$xpath)
```

5. Fuzzy versions of comparison operators for XPath expressions have been defined as XQuery functions. Similarly to *whereF*, comparison operators have been adapted to handle the RSVs:

```
declare function f:equalF($left,$right)
declare function f:lessF($left,$right)
declare function f:greaterF($left,$right)
```

3.2 Implementation of the Library

In order to implement our library in XQuery we have used the *XQuery Module* available in the *BaseX* processor [17]. In particular, we make use of the function *eval* that makes possible the manipulation of XPath expressions. This function is also available for *Exist* [20] and *Saxon* [18] processors. For instance, *down* is defined as follows:

```
declare function f:down($nodes,$query, $down){
  let $docDown := document{f:down_aux($nodes/*,$down,(),())}
  let $docQ := xquery:eval(concat('$x',$query), map { '$x' :=$docDown})
  let $docL := xquery:eval(concat('$x',$query), map { '$x' :=$nodes})
  return f:putListRSV($docL,f:getListRSV($docQ))
};
```

deep is defined as follows:

```
declare function f:deep($doc as node()*, $query, $deep as xs:double){
  let $docDeep := document{f:deep_aux($doc/*,$deep,1)}
  let $docQ := xquery:eval(concat('$x',$query), map { '$x' :=$docDeep})
  let $docL := xquery:eval(concat('$x',$query), map { '$x' :=$doc})
  return f:putListRSV($docL,f:getListRSV($docQ))
};
```

and *deep_ down* is defined as follows:

```
declare function f:deep_down($nodes as node()*,$query, $deep as xs:double, $down as
    xs:double){
  let $docDown := document{f:down_aux($nodes/*,$down,(),())}
  let $docDeep := document{f:deep_aux($docDown/*,$deep,1)}
  let $docQ := xquery:eval(concat('$x',$query), map { '$x' :=$docDeep})
  let $docL := xquery:eval(concat('$x',$query), map { '$x' :=$nodes})
  return f:putListRSV($docL,f:getListRSV($docQ))
};
```

Each fuzzy operator has been defined as a function, for instance, *and* (*Product* logic), *or+* (*Gödel* logic), *avg*, and *avg{a,b}* are defined as follows:

```
declare function f:andP($cond1,$cond2)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return $tv1*$tv2
};
declare function f:orG($cond1,$cond2)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return
    if ($tv1 > $tv2) then $tv1
    else $tv2
};
declare function f:avg($cond1,$cond2)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return (xs:double($tv1)+xs:double($tv2)) div (2)
};

declare function f:avg_ab($cond1,$cond2, $a, $b)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return (xs:double($tv1)*$a+xs:double($tv2)*$b) div ($a+$b)
};
```

4 Examples of Fuzzy XPath in XQuery

Now, we show how the previous fuzzy XPath queries can be written in XQuery. Let us now suppose the following fuzzy XPath query:

$$<< /hotels/hotel[[DOWN = 0.9]close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

We can now write the same query in XQuery as follows:

```
for $x in doc('hotels.xml')/hotels/hotel
let $y := f:whereF($x,f:equalF(f:down($x,'/close_to',0.9),'Sol'))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

We can see that fuzzy XPath expressions are written as XQuery expressions. This is the same kind of transformation from crisp XPath to XQuery. For instance:

$$<< /hotels/hotel[close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

can be translated into:

```
for $x in doc("hotels.xml")/hotels/hotel
where $x/close_to/text()="Sol"
return $x/@name
```

In the fuzzy case, *“=”* is transformed into *equalF*, and *where* as well as *return* become XQuery functions, with an extra argument to represent the context node. The query makes use of the function *down* of the library to compute the RSVs associated to *close_to*. In addition, the attribute *rsv*, which has been (internally) added to the output document, can be handled to show the answer in a sorted way, and even to define a threshold. Let us now consider the following query, that uses *deep* and *down*:

$$<< /hotels/hotel[[DEEP = 0.5; DOWN = 0.9]//close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

We can now write the same query in XQuery using the function *deep_down*:

```
for $x in doc('hotels.xml')/hotels/hotel
let $y :=
  f:whereF($x, f:equalF(f:deep_down($x,'//close_to',0.5,0.9),'Callao'))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

Let us now suppose the following fuzzy XPath expression that makes use of the *avg* operator.

$$<< //hotel[services/pool\ avg\ services/metro]/@name >>$$

Here, we use the function *avg* of the library, having as parameters both sides of the fuzzy condition:

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x, f:avg($x/services/pool,$x/services/metro))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

The same can be said for the following query, using $avg\{a,b\}$ having as parameters a and b.

$$<< //hotel[services/pool\ avg\{1,2\}\ services/metro]/@name >>$$

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x,f:avg_ab($x/services/pool, $x/services/metro,1,2))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

Let us now suppose the following queries that combine *deep* and *avg*, and *deep* and *and+*, respectively:

$$<< //hotel[[DEEP = 0.8]//close\_to/text() = \text{“}Sol\text{”}avg\{1,2\}\ //price/text() < 150]/@name >>$$

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x, f:avg_ab(f:equalF(f:deep($x,'//close_to',0.8),'Sol'),
      $x//price/text()<150,1,2))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

$$<< //hotel[([DEEP = 0.5]//close\_to/text() = \text{”}GranVia\text{”})\ and+(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$$

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x,f:andG(f:equalF(f:deep($x,'//close_to',0.5),'GranVia'),
      f:avg_ab($x//pool,$x//metro<200,3,2)))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

4.1 Benchmarks

Now, we would like to show the benchmarks we have obtained using our library. We have tested our library using data sets of different sizes. We have used as data sets traces of execution of MALP programs developed under our FLOPER tool. The FLOPER tool generates traces in XML format, with a high degree of tag nesting when a recursive program is executed. These data sets facilitate the testing of our structural based operators *deep* and *down*.

Fig. 4. Benchmarks

Query	*16Kb*	*700Kb*	*4.8Mb*	*15.4Mb*
Examined nodes in Q1	*28*	*148*	*298*	*448*
Examined nodes in Q2	*25*	*145*	*295*	*445*
Tree Depth	*21*	*101*	*201*	*301*
Q1	7.09 ms	25.47 ms	123.66 ms	461.6 ms
down in Q1	12.52 ms	107.1 ms	481.24 ms	2853.36 ms
deep in Q1	10.08 ms	74.17 ms	510.74 ms	1953.31 ms
deep and down in Q1	69.97 ms	102.0 ms	685.87 ms	7315.59 ms
Q2	5.77 ms	57.96 ms	172.03 ms	529.18 ms
avg in Q2	36.59 ms	1266.99 ms	9729.49 ms	60426.28 ms

In Figure 4 we can see the results, where we indicate the number of nodes examined in each tree, as well as the depth of the tree. We have compared the execution times for two XPath expressions in crisp and fuzzy versions. The first query is Q1:

$$<< //node/goal >>$$

and the second query is Q2:

$$<< //node[goal[contains(text(),"p(")]\ and\ substitution[contains(text(),"g(")]]//goal >>$$

We have used the BaseX Query processor in a Intel Core 2 Duo 2.66 GHz Mac OS machine.

5 Related Work

Fuzzy logic plays a key role in information retrieval and the need for providing fuzzy/flexible mechanisms to XML querying has recently motivated the investigation of extensions of the XQuery/XPath language. We can distinguish those in which the main goal is the introduction of fuzzy information in data (similarity, proximity, vagueness, etc) [25, 26, 8, 7, 15, 27, 23, 24] and the proposals in which the main goal is the handling of crisp information by fuzzy concepts [16, 9, 10, 13, 12, 19, 11]. Our work focuses on the second line of research.

Fuzzy versions of XQuery have been previously studied in some works. The closest to our approach is [16], in which preferences can be described by queries in order to retrieve discriminated answers by user's preferences. FLOWR expressions are extended to cover with fuzzy values and answers. The main aim of their work is to extend XQuery with definition of fuzzy terms: good, cheap, high, young, etc., defined as fuzzy predicates that can be imposed in XPath expressions. They extend XQuery datatypes with *xs:truth* and incorporate *xml:truth* as attribute to represent degree of satisfaction. Nevertheless, they lack on an implementation, and therefore we cannot compare our proposal with them, although we believe that a similar technique we have proposed here can be used. In [25], they also extends the syntax of XQuery, in particular, the expression *where* to cover with priority and thresholding. Their approach is focused on querying fuzzy XML data, and therefore their proposal is different from our. The have developed an implementation using Java on top of the Exist [20] XQuery processor. A fuzzy query

is transformed into standard XQuery to be executed. Fuzzy data querying is also the main aim of the work of [26], in which they propose a fuzzy XML Shema and algebraic operators to handle fuzzy data over an schema. They provide transformations from the algebraic operators to XQuery (and XPath) expressions. Again, their approach is different from our, since they work with fuzzy XML data as input.

Fuzzy versions of XPath have been previously studied in some works. The closest works to our proposal are [9, 10] in which authors introduce in XPath flexible matching by means of fuzzy constraints called *close* and *similar* for node content, together with *below* and *near* for path structure. In addition, they have studied the *deep-similar* notion for tree matching, and fuzzy versions for *not*, *and* and *or* operators. In order to provide ranked answers they assign a RSV to each item. Our work is similar to the proposed by [9, 10]. The *below* operator of [9, 10] is equivalent to our proposed *down*: both extract elements that are direct descendants of the current node, and the penalization is proportional to the distance. The *near* operator of [9, 10], which is defined as a generalization of *below*, ranks answers depending on the distance to the required node, in any XPath axis. Our proposed *deep* ranks answers depending of the distance to the current node, but the considered nodes can be direct and non direct descendants. Therefore our proposed *deep* combined with *down* is a particular case of *near*. To have the same expressive power as *near* we could incorporate to our framework a new operator to rank answers from bottom to up. With respect to *similar* and *close* operators proposed in [9, 10], our framework lacks similarity relations and rather focuses on structural (i.e. path-based) flexibility.

In [13], the authors propose to give a satisfaction degree to XPath expressions based on associating weights to XPath steps. Relaxing XPath expressions when the path does not match the XML schema is the main goal of this work. They have studied how to compute the best k answers. In this line, in [12, 14] XPath relaxation is studied given some rules for query rewriting: axis relaxation, step deletion and step cloning, among others. The proposed *deep-similar* notion of [9, 10] can be also considered a relaxation technique of XML tree equality. Our work has some similarities with these proposals: *deep* and *down*, and also the use of *avg* operator, are mechanisms for relaxing queries and giving priority to paths and answers. We have also studied in [3] how to introduce axis relaxation, step deletion and step cloning in our approach, but the proposed implementation does not still include these mechanisms. It is considered as future work.

6 Conclusions and Future Work

In this paper we have presented an XQuery based implementation of a fuzzy version of XPath. Fuzzy XPath incorporates mechanisms to rank answers depending on the location of the item in the XML tree of input, as well as to give priority to queries. The output of a query contains an RSV in each item according to the user's preferences. We have described the elements of the XQuery library that make possible to express fuzzy queries against crisp XML data. As future work, we plan the following steps. Firstly, to incorporate new mechanisms of searching and ranking to queries. We have previously studied [3, 4] how to penalize answers when a given XPath expression is incorrect, and tags have to be jumped, switched and added. We believe that these mechanisms can be implemented also in XQuery. Secondly, we would like to extend our work to other fuzzy logic mechanism (vagueness, similarity, etc). Another direction

we can take is to introduce the operators in XQuery and then use a parser to translate the user expression in standard XQuery language. Finally, we would like to improve the performance of our implementation, for instance, thanks to the use of thresholding, following [5]. Up to now, thresholding is achieved on the output of the query, and dynamic thresholding would improve the performance.

References

1. Jesús Manuel Almendros-Jiménez, Alejandro Luna, and Ginés Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electr. Notes Theor. Comput. Sci.*, 282:3–18, 2012.
2. J.M. Almendros-Jiménez, A. Luna, and G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11*, pages 186–193. Springer Verlag, LNCS 6826, 2011.
3. J.M. Almendros-Jiménez, A. Luna, and G. Moreno. A XPath Debugger Based on Fuzzy Chance Degrees. In P. Herrero et al., editor, *On the Move to Meaningful Internet Systems: Proceedings OTM 2012 Workshops, Rome, Italy, September 10-14*, pages 669–672. Springer Verlag, LNCS 7567, 2012.
4. J.M. Almendros-Jiménez, A. Luna, and G. Moreno. Annotating Fuzzy Chance Degrees when Debugging Xpath Queries. In *Proc. of the 12th International Work-Conference on Artificial Neural Networks, IWANN'13*, pages 300–311. Springer Verlag, LNCS 7903, Part II, 2013.
5. J.M. Almendros-Jiménez, A. Luna, and G. Moreno. Dynamic Filtering of Ranked Answers When Evaluating Fuzzy XPath Queries. In *Rough Sets and Current Trends in Soft Computing 2014*, pages 319–330. Springer Verlag, LNAI 8536, 2014.
6. A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0. *W3C*, 2007.
7. P. Buche, J. Dibie-Barthélemy, O. Haemmerlé, and G. Hignette. Fuzzy semantic tagging and flexible querying of XML documents extracted from the Web. *Journal of Intelligent Information Systems*, 26(1):25–40, 2006.
8. P. Buche, J. Dibie-Barthélemy, and F. Wattez. Approximate querying of XML fuzzy data. *Flexible Query Answering Systems*, pages 26–38, 2006.
9. A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini. A fuzzy extension of the XPath query language. *Journal of Intelligent Information Systems*, 33(3):285–305, 2009.
10. E. Damiani, S. Marrara, and G. Pasi. FuzzyXPath: Using fuzzy logic and IR features to approximately query XML documents. *Foundations of Fuzzy Logic and Soft Computing*, pages 199–208, 2007.
11. E. Damiani, S. Marrara, and G. Pasi. A flexible extension of XPath to improve XML querying. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 849–850. ACM, 2008.
12. B. Fazzinga, S. Flesca, and F. Furfaro. On the expressiveness of generalization rules for XPath query relaxation. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, pages 157–168. ACM, 2010.
13. B. Fazzinga, S. Flesca, and A. Pugliese. Top-k Answers to Fuzzy XPath Queries. In *Database and Expert Systems Applications*, pages 822–829. Springer, 2009.
14. Bettina Fazzinga, Sergio Flesca, and Filippo Furfaro. Xpath query relaxation through rewriting rules. *Knowledge and Data Engineering, IEEE Transactions on*, 23(10):1583–1600, 2011.

15. A. Gaurav and R. Alhajj. Incorporating fuzziness in XML and mapping fuzzy relational data into fuzzy XML. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 456–460. ACM, 2006.
16. Marlene Goncalves and Leonid Tineo. Fuzzy XQuery. In *Soft Computing in XML Data Management*, pages 133–163. Springer, 2010.
17. Christian Grün. BaseX. The XML Database, 2014. `http://basex.org`.
18. M. Kay and S. Limited. Ten reasons why Saxon XQuery is fast. *IEEE Data Engineering Bulletin*, 1990.
19. H.G. Li, S.A. Aghili, D. Agrawal, and A. El Abbadi. FLUX: fuzzy content and structure matching of XML range queries. In *Proceedings of the 15th international conference on World Wide Web*, pages 1081–1082. ACM, 2006.
20. Wolfgang Meier. eXist: An open source native XML database. In *Web, Web-Services, and Database Systems*, pages 169–183. Springer, 2003.
21. P.J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In Nick Bassiliades et al., editor, *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*, pages 119–126. Springer Verlag, LNCS 3521, 2008.
22. P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In M. Dean et al., editor, *Proc. of 4nd Intl Symposium on Rule Interchange and Applications, RuleML'10*, pages 20–34. Springer Verlag, LNCS 6403, 2010.
23. B. Oliboni and G. Pozzani. Representing fuzzy information by using XML schema. In *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*, pages 683–687. IEEE, 2008.
24. B. Oliboni and G. Pozzani. An XML Schema for Managing Fuzzy Documents. *Soft Computing in XML Data Management*, pages 3–34, 2010.
25. P Ueng and S Skrbic. Implementing xquery fuzzy extensions using a native xml database. In *Computational Intelligence and Informatics (CINTI), 2012 IEEE 13th International Symposium on*, pages 305–309. IEEE, 2012.
26. Ekin Üstünkaya, Adnan Yazici, and Roy George. Fuzzy data representation and querying in xml database. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 15(supp01):43–57, 2007.
27. L. Yan, ZM Ma, and J. Liu. Fuzzy data modeling based on XML schema. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1563–1567. ACM, 2009.