A Fuzzy Extension for the XPath Query Language

Alessandro Campi[1], Ernesto Damiani[2], Sam Guinea[1],
Stefania Marrara[2], Gabriella Pasi[3], and Paola Spoletini[1]

(1)Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. da Vinci 32, I-20133 Milano, Italy
campi|guinea|spoleti@elet.polimi.it

(2)Università degli Studi di Milano
Dipartimento di Tecnologie dell'Informazione
via Bramante 65 26013 Crema (CR), Italy
damiani|marrara@dti.unimi.it

(3) Università degli Studi di Milano Bicocca
DISCO
Viale Sarca 336 20126 Milano (MI), Italy
pasi@disco.unimib.it

Abstract XML has become a widespread format for data exchange over the Internet. The current state of the art in querying XML data is represented by XPath and XQuery, both of which rely on Boolean conditions for node selection. Boolean selection is too restrictive when users do not use or even know the data structure precisely, e.g. when queries are written based on a summary rather than on a precise representation of the schema. In this paper we describe a XML querying framework, called FuzzyXPath, based on Fuzzy Set Theory, relying on fuzzy conditions for the definition of flexible constraints on stored data. To this end, we introduce a function called "deep-similar", which aims at substituting XPath's typical "deep-equal" function. Its goal is to provide a degree of similarity between two XML trees, assessing whether they are similar both structure-wise and content-wise. Several query examples are discussed in the field of XML-based metadata for e-learning.

1 Introduction

In the last ten years, the need for expressing in a declarative form complex manipulations of XML trees has raised a lot of interest on XML query languages. The World Wide Web Consortium (W3C) has defined two standard languages for querying XML data: XPath and XQuery. XPath allows selecting XML node sets via *tree traversal* expressions. Although XPath is not a fully-fledged query language, it is expressive enough for many practical tasks, and has been adopted within XQuery for expressing selection conditions. XQuery adds to XPath the capability of working on multiple XML documents, of joining results, and of transforming and creating XML structures. XPath selection, used by both languages, is Boolean in nature: it partitions XML nodes into those which fully satisfy the selection condition, and those which do not. However, Boolean conditions can be —in some scenarios— not flexible enough for effectively querying XML data. A few considerations can be made to justify this claim. The first

is about prescriptive schemas defining the tree structure of individual documents. Even when XML schemas do exist, they may be not available to users. Moreover document trees with the same schema may be widely different (both in used tags and nesting), and hence the schema will allow for diverse instantiations, making it difficult to predict a particular document structure from the schema. As a consequence, users often end up defining *blind* queries, i.e. queries written without a precise knowledge of the schema. This happens either because users do not know the XML schema in detail, or because they do not know exactly what they are looking for. Finally, the same XML tree can be sometimes described using different schemas. These observations are highly relevant for practical applications; for instance, they are all well-known consequences of providing users with schema summaries rather than with complete XML schemas [1].

In order to fix our ideas we shall focus on the field of e-learning, which will constitute our explanatory context throughout this paper. We have chosen to use learning objects as our example because they are typically described using XML metadata that contain many different kinds of data. *Learning Management Systems* (LMSs) consist of distributed systems through which an institute can provide personalized e-learning content to its students. The main component of such systems is the repository, which contains information concerning (a) the system's end-users (i.e., teachers and students), (b) the electronic content items (called *learning objects*) (LO) created by the teachers and consumed by students, and (c) other physical content (e.g., books) the student can consult for more information. Data describing LOs are called LO metadata (LOM). A simplified representation of the structure of the data contained in an e-learning repository is shown in Figure 1. We shall use the situation depicted in Figure 1 as a running example throughout the paper.

The data structure used for representing a student comprises identification fields such as the student's name and surname, and her/his unique identification number. Each student has also a curriculum vitae field. In our example, the students are not required to use a common structure for their curricula. We will assume that all the curricula will provide at least some of, and probably not all, the following information: previous studies, known languages, previous jobs, professional training, etc. All these data are used by the LMS to personalize the student's program, by suggesting the learning objects (LOs) and books that are most appropriate for her/him. In this field of research it is common to describe electronic content, i.e., learning objects (LOs), using the LOM standard[1] and its XML representation. The choice of this running example is motivated by the fact that LOMs data management poses all the challenges of XML data management: LOMs contain all data types (numerical, string,...) and the complexity of structures involved is variable and covers most typical scenarios. Indeed LOMs lie on a middle-ground between document-centric and data-centric XML: both these XML document categories have been widely studied by the Information Retrieval (XML retrieval) and database

[1] This IEEE standard specifies the syntax and semantics of Learning Object Metadata, defined as the attributes required to describe a Learning Object. The Learning Object Metadata standard focuses on the minimal set of attributes needed to allow these Learning Objects to be managed, located, and evaluated. Relevant attributes of Learning Objects include object type, author, owner, terms of distribution, format and pedagogical attributes such as teaching or interaction style, grade level, mastery level, and prerequisites.

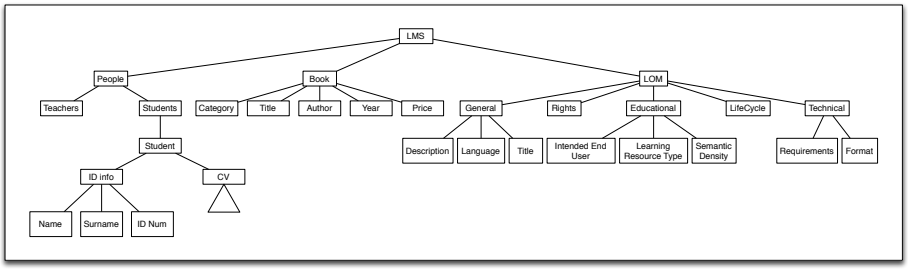

Figure 1. A simplified representation of the structure of the data contained in the LMS.

communities. Instead LOMs are a good example of an important category of data posing problems which have not been fully addressed by current techniques. Moreover, the structures that are used are sufficiently diverse to be significant.
In our example we assume that the system is distributed across multiple repositories, where the same content may be duplicated and stored with slightly different metadata.
We shall use the above scenario to illustrate a framework for querying XML data that goes beyond Boolean selection, and that allows the user to define 'vague' and flexible selection conditions in order to obtain ranked results according to her/his preferences. To do so, we use concepts coming from the area of Fuzzy Set Theory. The main idea is that the constraint evaluation produces a fuzzy subset, with the consequence of associating a numeric value (the membership degree) with each information item. This represents the Retrieval Status Value (RSV) of the associated item. To obtain these gradual results fuzzy conditions are allowed, and expressed by linguistic labels with a membership function associated. Concretely, we propose FuzzyXPath[2] an extension of XPath query language accommodating fuzzy selection. We describe our extension with respect to XPath since it presents a simpler starting point with respect to XQuery, but our entire approach is constructed with the aim to be easily applicable to XQuery. Extensions can be summarized as follows:

- Fuzzy conditions: the user can express vague selection conditions formally represented as fuzzy subsets on the attribute domain.
- Fuzzy Tree Matching: Standard XPath provides a *deep-equal* function that can be used to assess whether two sequences contain items that are atomic values and are equal, or that are nodes of the same kind, with the same name, whose children are deep-equal. This can be restrictive, so we propose an extension named *deep-similar* to assess the structure similarity.

With respect to an earlier description of the approach[2], in this paper we go into the details of fuzzy conditions syntax and semantics, we add the fuzzy condition APPROXIMATELY and present a query evaluation proposal based on the tree edit distance. In our approach the relevance estimate is based on the evaluation of *fuzzy constraints* on crisp data. A fragment obtains a higher evaluation score with respect to another one if

[2] Since *FXPath* has recently been used with the meaning of *Functional XPath*, in our work we suggest the name *FuzzyXPath* to avoid misunderstanding.

it satisfies more the fuzzy constraints contained in the query and appropriately combined. As a consequence the ranking of the fragments is ”well founded” if the fuzzy constraints are ”well defined”.
When FuzzyXPath query results are returned, they are associated with a ranking value (the *Retrieval Status Value — RSV*) indicating “how much” each data item satisfies the selection condition. For example, while searching for LOs published in a year around 2000, we might retrieve LOs published in 2000, in 2001, 2002, with distinct RSV indicating the estimated relevance with respect to users’ preferences. Returned items are wrapped into annotations, so as to recall XML tagging:

```
<!--RankingDirective RankingValue="1.0" -->
  <LO year="2000"> <title>t1</title> </LO>
<!-- /RankingDirective -->
<!-- RankingDirective RankingValue=".8" -->
  <LO year="2001"> <title>t2</title> </LO>
<!-- /RankingDirective -->
<!-- RankingDirective RankingValue=".65" -->
  <LO year="2002"> <title>t3</title> </LO>
<!-- /RankingDirective -->
```

In this example it is possible to notice the presence of a ranking directive containing the ranking value —in the set $[0, 1]$— of each retrieved item. The closer it is to 1, the better the item satisfies the condition (i.e. to be published in a year near 2000). This way, large result sets can be organized according to user’s interests.
The rest of this paper is structured as follows. Section 2 presents relevant related work. Section 3 provides some preliminary concepts useful to the comprehension of the proposed approach; Section 4 presents the application of fuzzy conditions useful to extend XPath regarding both the node contents and names. Section 5 illustrates the weighting of the document tree, it discusses vagueness in the tree structure, fuzzy tree matching and our implementation of the “deep-similar” function. Section 6 brings the two approaches together, Section 7 presents a prototype tool and finally Section 8 draws our conclusions.

2 Related Work

In this section we briefly review the main approaches to flexible querying techniques focusing on applications of of Fuzzy Set Theory.
A stream of research on *Fuzzy Pattern Matching* (FPM) was started in the Eighties. Given a query in which fuzzy selection conditions are expressed, and a database containing imprecise or vague attribute values, FPM returns two matching degrees (degrees of possibility and necessity)[3].
The work [4] proposes a counterpart to FPM, called *Qualitative Pattern Matching* (QPM), for estimating levels of matching between a request and data expressed by words. Given a request, QPM rank-orders the items which possibly, or which certainly match the requirements, according to the preferences of the user.
As far as flexible database querying is concerned, fuzzy quantifiers allowing to define aggregated concepts have been described in [5] and [6]. The seminal paper [7] showed a

convenient way to model flexible queries using fuzzy sets. Some of the above mentioned approaches deal with fuzzy data in datasets while the approach presented in this paper is aimed at adding flexibility in query formulation while having only crisp data in the datasets.

Many attempts to extend relational query languages like SQL with fuzzy capabilities were undertaken in recent years. For instance, [8] describes *SQLf*, a language that extends SQL by introducing fuzzy conditions evaluated on crisp information. The *FSQL* system [9], developed upon Oracle, represents imprecise information as possibility distributions stored in standard tables. FSQL queries are translated into ordinary SQL queries which call special functions to compute the degrees of matching.

The approach presented in [10] applies fuzzy logic to multimedia databases where documents have to be retrieved and selected depending not only on their contents, but also on the idea the user has of their appearance, through queries specified in terms of user criteria.

As far as semi-structured information processing is concerned, several research approaches have been proposed to enhance flexibility of XML querying. An early approach [11] was based on fuzzy encoding of XML data trees. W.r.t. our approach, that method did not attempt to define a query language; rather, it supported introducing new nodes in the XML tree structure in order to carry out fuzzy similarity comparison of XML data trees. This approach was later extended in [12] providing some flexibility in content comparison with the concept of XML data *smushing*. A later paper [13] proposed an approach based on XML query rewriting, supporting renaming and deletion of nodes in the query. Hybrid techniques [14] have also been proposed, where XML data are encoded and queries are rewritten. On the one hand, hybrid techniques can provide an accurate computation of the query cost; on the other hand, it is very difficult to implement them because they require ad hoc XML data indexing.

A recent approach to this problem [15] proposes a dynamic summarization and indexing method, FLUX, based on Bloom filters and B^+-trees. Also, the work [16] presents an indexing method to execute approximate queries on XML documents taking into account approximation on both document structure and content. The proposed indexing aims to reduce the complexity of finding approximate query patterns, avoiding sequentially scanning of all documents in the collection.

Another recent paper [17] proposes a fuzzy-based XML querying system that performs approximate comparisons between query and data trees. This technique supports imprecise data via possibility distributions. The authors claim that their system is fully compatible with XML querying standards since the final rewriting is performed in XQuery; however, their query rewriting is based on a *mediated architecture* called *MIEL++* that requires several rewriting steps; the number of rewritings can be controlled by a threshold on the maximal transformation cost and the parameters associated with the views which determine the number of deletions and renaming which are authorized.

In [18] the authors propose an approach to approximate query answering in which, instead of working directly on the data, they apply the structural component of the query to execute a reworking of the documents' schemas by means of a *schema matching* process.

The closest approach to ours is the work [19] which presents *FlexPath*, an attempt to integrate database-style query languages such as XPath and XQuery and full-text search on textual content. FlexPath considers queries on structure as a template, and looks for answers that best match this template and the full-text search. To achieve this, FlexPath provides an elegant definition of relaxation on structure and defines primitive operators to span the space of relaxations. Query answering is now based on ranking potential answers on structural and full-text search conditions. However, FlexPath does not provide a syntax for presenting flexible structural requirements both for the inquired documents and for the retrieved answers.
Finally we mention the XQuery 1.0 and XPath 2.0 Full-text (XQFT) language, which has been developed by the W3C to extend XQuery and XPath with full-text search capabilities. XQFT allows users to specify a mix of structured and complex full-text predicates, and also allows users to score/rank such queries. Another text retrieval based approach is NEXI, which is used in the INEX evaluation of XML retrieval. While they concentrate on flexibility in text search, our approach introduces flexibility in the query expression by means of fuzzy constraints, and proposes a methodology to compute the similarity between query and data trees.

3 Technical Background

In this section we present the background of our work. We briefly recall the semantics of XPath queries [20] and the main definitions of Fuzzy Set Theory [21]. These represent the basis of our approach and their introduction allows a better understanding of our proposal.

3.1 XPath

The XML Path Language (XPath) is an expression language for extracting portions of or computing information on XML documents by exploiting the tree representation of XML documents, it provides the ability to navigate trees and select nodes using a variety of specified selection methods.
XPath uses a declarative notation and allows to extract information through path expressions. Each expression is composed by a finite sequence of steps. A step is composed by three main elements: an *axis specifier*, that indicates the navigation direction within XML tree representation, a *node test*, that specifies a node name or, more in general, an expression, which allow to identify one or more particular nodes or paths in the specified direction, and a *predicate*, that is an expression of any complexity, which must be satisfied before the preceding node will be matched by an XPath expression.
Each path expression describes the types of nodes to match based on the hierarchical relationship between the nodes. For example, the expression `LO/Educational` means finding all `Educational` elements contained within `LO` elements. This notation enables to query an XML tree as a hierarchy of nodes. The result of the evaluation of an expression is the set of all `Educational` elements returned in the order they occurred in the document. At each step in the traversal, nodes selected in that step can be filtered using a predicate, also called qualifier. As an example, if we

ask for `/LO/Educational/LearningResourceType[source]` the result is *all* `LearningResourceType` elements that have at least one child element named `source`. Previous examples are all *absolute* XPath expressions (since they involve a leading "/"). The general meaning of an expression is defined relatively to a context node in the tree. Starting from a particular context node in the tree, each other node can be reached by means of sequence of axis specifiers.

3.2 Fuzzy Set Theory

Fuzzy Set Theory was introduced to extend classical Set Theory, in which the membership of an element to a set is binary, i.e. an element either belongs or does not belong to a set. Differently, fuzzy set theory allows to assign a degree of membership of each element to a given set. More precisely the membership of an element to a set is not defined in a binary way, but is defined by means of a *membership function*. Given a domain X, more often called universe, a membership function $\mu_f : X \rightarrow [0, 1]$ assigns to each object $x \in X$ a membership degree $\mu_f(x) \in [0, 1]$. As a consequence of this new definition of membership, the classical set operations, such as intersection, union and complement were redefined[3].
As a consequence, the classical Boolean operators, $\wedge$, $\vee$ and $\neg$, are re-defined using the redefined set operations. In this work we adopt the following definitions:

$$\mu_{A \wedge B} = min\{\mu_A(x), \mu_B(x)\};$$
$$\mu_{A \vee B} = max\{\mu_A(x), \mu_B(x)\};$$
$$\mu_{\neg A} = 1 - \mu_A;$$

4 A Fuzzy Extension of XPath

In this section we describe an extension to the XPath language and its semantics. We propose to allow fuzzy conditions to express flexible constraints in query formulation. Fuzzy conditions are expressed by using linguistic labels identifying fuzzy sets (Section 4.1) and allowing flexible specification for the axes (Section 4.2). Moreover, we define a flexible tree matching algorithm, to exploit flexibility also in evaluating similarity between tree structures (Section 4.3).

4.1 Supporting flexibility in value matching

In order to allow the query constraints to express vagueness, we introduce two fuzzy constraints, namely `CLOSE` and `SIMILAR`, that apply to specific items within XML documents and are associated with a coherent fuzzy subset. Constraints evaluation for a given item produces a numeric value which expresses the satisfaction degree of the constraint when applied to that particular item.

[3] By contrast with classical set theory, there are different possible definitions for these operations, although some of them are more often adopted. For an overview of these definitions refer to [21].

These two flexible constraints can be applied to different kinds of data and, as a consequence, are evaluated by means of different definitions. Indeed, the XML Schema standard supports more than 40 elementary data types, that, by definition, carry different kind of information. When the elementary type is structured, like numeric or date types, the definition of a fuzzy constraint to evaluate conditions on values is quite straightforward[4]. Here, for the sake of conciseness, we shall deal with three data types, namely text, integer and date.
Before giving the formal definition for each case, we briefly introduce their meaning and utility in querying XML documents.

`CLOSE` This linguistic label is used to specify a fuzzy condition in which closeness to the content value specified in the query must be evaluated against the values in the stored information items. This fuzzy condition relaxes the usual Boolean condition of exact matching to a precise value specified in the query. `CLOSE` is useful whenever the query involves a numeric value that does not need to be exact or when the value is a string that may contain typos and synonyms. As an example consider the number of slides in a presentation. When a user is looking for a LOM containing a presentation that is not too big, she can quantify this concept saying that the number of slides is *close to 100*. Consider another example: if a user queries the dataset asking for the LOMs in which the author is Betty Price it will find all the LOMs in which the name is exactly the one expressed in his/her query, but if he/she asks for the LOM whose author has a name "close" to Betty Price, the result will contain also the LOMs where the content of the author element is spelled incorrectly (for example where the author is equal to Betty Prize).

`SIMILAR` This linguistic label allows to specify in the query a flexible constraint to select nodes with a name similar to a given name, i.e. the node matching does not require a perfect matching. This is useful, because often XML tag or attribute names can be introduced with typos or substituted by a synonyms. Consider as an example the tag representing the author of a LOM. If a user queries the dataset asking for the LOMs whose author is Betty Price, using the `SIMILAR` construct you can find it in a tag called `writer` (where a synonym has been used).

Hence, the two selected labels allow to specify flexible constraints on crisp data and to take in consideration also data affected by typing errors.
We can now formally define the use of these labels. Their syntax is defined as follows.

Definition 1 *The constraint* `CLOSE` *is applied to content values. It requires to evaluate the similarity of stored (crisp) values with respect to a single value specified in the query. The syntax is:*

```
"[{ selection_node" ("NOT")? "CLOSE" "compare_value }]"
```

where `selection_node` *is a path expression representing the node to be retrieved the content of which is taken to assess its similarity to the value specified by the clause* `compare_value`. *The (optional) negation operator* `NOT` *is used to negate the similarity constraint* `CLOSE` *(as defined in 3.2).*

[4] Unfortunately, the same cannot be said for user-defined `ComplexType`s. The reader interested in fuzzy matchings for complex XML types is referred to [22]

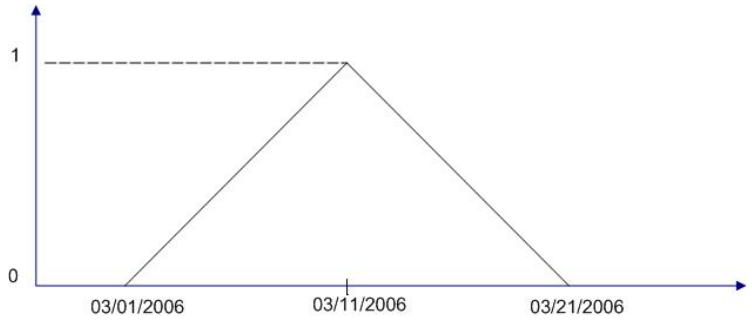

Figure 2. The `CLOSE` fuzzy constraint applied to a date of birth

Definition 2 *The constraint* `SIMILAR` *is applied to a tag or to an attribute name. It allows to specify in the query a constraint to select nodes with a name similar to a given name:*

```
"[{ selection_node" ("NOT")? "SIMILAR" "node_name }]"
```

where `selection_node` *is an XPath path representing the node to be retrieved whose tag name is expected similar to the name expressed by the clause* `node_name`. *The (optional) negation operator* `NOT` *is used to negate the similarity constraint* `CLOSE` *(as defined in 3.2).*

We now give a formal definition of the semantics of these fuzzy constraints. This is done by defining the membership functions associated with the allowed fuzzy constraint. The domains of reference (universe of discourse) are texts, integers and dates.

For integers and dates, we can define the fuzzy membership function as a symmetric triangular function centered in $x \in X$. Triangular functions have proved in literature to work well in text-retrieval contexts. Moreover there are several techniques to construct membership functions automatically [23], while other works have studied the best way to associate membership functions and data domains (as instance see [24]). In figure 2 an example of membership function for the `CLOSE` constraint applied on a date of birth is shown.

When the flexible constraints `CLOSE` and `SIMILAR` are applied to free text nodes, we consider two different approaches: the first analyzes the similarity between two strings, while the other performs a semantic analysis between words. To address the first problem we use *Levenshtein distance* [25]. Given two strings S and T, it performs all the possible matchings among their characters, obtaining a matrix $|S| * |T|$ from which the distance between the two words is obtained. This approach can be useful when the same word appears in different points written with different spellings, but in general it does not help us to identify two words with the same meaning but different

lexical stems. For analyzing the similarity from a semantics point of view it is necessary to integrate the system with a dictionary that contains all relevant synonyms [5].

4.2 Supporting flexibility in path structures

In order to provide a more flexible support to query process, we also define two flexible conditions for flexible matching of path structures. Namely, we introduce `BELOW` and `NEAR` that express axis specification in a fuzzy way. Informally, the fuzzy condition `BELOW`, inserted into a path expression, allows to extract elements, attributes or text that are direct descendants of the current node, giving a penalty which is proportional to the result's distance from the current node. `NEAR` is a generalization of `BELOW`. It is inserted into a path expression, the selection tree, to extract elements, attributes or text that are placed close to the current node, giving a penalty which is proportional to the result's distance from the current node. In this case the similarity evaluation is performed within any XPath axis.

Next we give an idea of possible uses of the proposed fuzzy constraint in order to show their utility.

`BELOW` Often, users expect to find the information they are looking for are in a particular node, while in reality it lies in a more deeply nested subtree. For example consider the query searching for students that have taken a course of "English" in their university career. The user could guess this information to be in node <`university-career`> but curricula could be structured more in detail, e.g. by putting language exams in a subsection. Students who have structured their curricula this way could be selected by a query using the `BELOW` fuzzy constraint.

`NEAR` The `NEAR` construct is motivated as a generalization of `BELOW`. In some cases a query searches information that would be significant even if it located in a node "close" to the target one. Let us consider again the curricula example. Querying for someone who has in her/his professional experiences a working holiday in a foreign country is not easy, because different curricula could use different nodes to store professional experiences (e.g., some distinguish work abroad from work within the country, and others do not). The application of `NEAR` fuzzy constraint in the query allows to obtain these last kind of curricula as a part of the query result.

The syntax of fuzzy structural constraints is formally defined as follows:

Definition 3 *Constraint* `BELOW`*, inserted into the axis of a path expression, allows to extract elements, attributes or text that are direct descendants of the current node, giving a penalty which is proportional to the result's distance from the current node. The syntax is:*

```
"[{ selection_node" ("NOT")? "BELOW" "node_name }]"
```

where `selection_node` *is an XPath node name representing the node to be retrieved as descendant node of the node expressed by the clause* `node_name`*. The (optional) negation operator* `NOT` *is used to negate the similarity flexible constraint* `CLOSE` *(as defined in 3.2).*

[5] The vocabulary we use is called *JWordNet*, and is available at `http://wordnet.princeton.edu/`.

Definition 4 *Constraint* NEAR*, inserted into the axis of a path expression, allows to extract elements, attributes or text that are successors of the current node, giving a penalty which is proportional to the result's distance from the current node. The syntax is:*

```
"[{ selection_node" ("NOT")? "NEAR" "node_name }]"
```

where selection_node *is an XPath node name representing the node to be retrieved whose path position can differ from the position of the node expressed by the clause* node_name*. The (optional) negation operator* NOT *is used to negate the similarity flexible constraint* CLOSE *(as defined in 3.2).*

As an example, consider the following query:

```
\LO\{NEAR::}duration
```

Here, we search for a duration element placed near a LO element. The degree of satisfaction of this constraint is a function of the number of steps needed to reach the duration element starting from the LO one.
Besides the previous flexible constraints, we introduce another derived one, called APPROXIMATELY, related to fan-out. It allows to select the elements with a given name that have a number of direct descendants close to the one indicated in the query. APPROXIMATELY is a derived constraint that can be substituted by a CLOSE constraint applied to the result of the *count* function on the sons of a given node.
As an example, if we suppose that the teachers element contains one ore more teacher, where each teacher contains the list of courses (one ore more) where he/she teaches, we can use the query

```
\teachers\teacher{APPROXIMATELY[3]::}course
```

to extract a ranked list of teachers whose top results are teachers that have 3 (or a number close to 3) courses.

4.3 Supporting flexibility in matching tree structures

Finally we present a possible proposal to support flexible tree matching. The notation we propose of partial matching is based on the *deep-similar* function, allowing to compute the distance between two XML trees based on the concept of *Tree Edit Distance* (TED), a well-known approach for calculating how much it costs to transform one tree (the *source* tree) into another (the *destination* tree). This approach has been widely used to evaluate structural similarity in XML documents as shown in [26], to correlate XML data streams as shown in [27], or, in a field close to XML, for the automatic web news extraction as shown in [28]. Generally speaking, we rely on deep-similarity to assess the similarity between two arguments of a function appearing in a FuzzyXPath query. *deep-similar* can be considered as the fuzzy extension of the *deep-equals* function. Before applying the *deep-similar* function, it is important to clarify when it will be used during FuzzyPath query evaluation. In FuzzyXPath all path expressions appearing within a deep-similar comparison[6] correspond to the sub-trees rooted in each

[6] Paths appearing elsewhere do maintain the usual node-set semantics.

of the nodes in the target node-set. In other words, for each selection path featured in a comparison we take the trees rooted in each of the nodes obtained interpreting the path as if it were an ordinary XPath, and then apply deep-similarity to them[7] .
The formal definition of *deep-similar* is given below:

Definition 5 (Deep-similar) *Given two XML trees T_1 and T_2, deep-similar(T_1, T_2) is the function that returns their degree of similarity as a value in [0,1]. Given C as the cost of transforming T_1 into T_2 using Tree Edit operations,$C \in [0, 1]$, this degree of similarity is given as k^C, where $k \in (0, 1)$. Therefore, if two trees are completely different, their degree of similarity is small (inversely proportional to the total cost of the Tree Edit Operations); if they are exactly the same —both structure-wise and content-wise— their degree of similarity is 1.*

In order to transform the XML tree T_1 into T_2, our deep-similar function uses the following operations:

Definition 6 (Insert) *Given a XML tree T, a XML node n, a location loc (defined through a path expression that selects a single node p in T), and an integer i, $Insert(T, n, loc, i)$ transforms T into a new tree T' in which node n is added to the first level children nodes of p in position i.*

Definition 7 (Delete) *Given a XML tree T, and a location loc (defined through a path expression that selects a single node p in T), $Delete(T, loc)$ transforms$T$ into a new tree T' where node n is removed.*

Definition 8 (Modify) *Given a XML tree T, and a location loc (defined through a path expression that selects a single node p in T), and a new value v, $Modify(T, loc, v)$ transforms T into a new tree T' where the content of node p is replaced by v.*

Definition 9 (Permute) *Given an XML tree T, a location loc_1 (defined through a path expression that selects a single node n_1 in T), and a location loc_2 (defined through a path expression that selects a single node n_2 in T), $Permute(T, loc_1, loc_2)$ transforms T into a new tree T' in which the locations of nodes n_1 and n_2 are exchanged.*

Before these operations can be applied to the source tree, it is necessary to perform a preprocessing of the XML document divided in two steps. The first step consists in weighing the nodes within the two XML trees following the approach described in Section 5, in order to discover the importance they have within the trees. This operation will provide the basis to calculate the cost of each tree transformation. The second consists in matching the nodes in the source tree with those in the destination tree. This step is necessary to outline which nodes will be target of the different Tree Edit operations. The costs of these Tree Edit Operations will be presented as soon as we complete a more in depth explanation of the two aforementioned steps.

[7] Of course, this turns comparing two paths S and T into computing *deep-similarity* of multiple pairs of subtrees (potentially as many as the product of S and T node sets cardinalities) and taking the maximum (coherently with the typical existential quantification of XPath conditions) of the resulting similarity values. We shall not further elaborate on this issue, as it does not impact on the *deep-similarity* semantics.

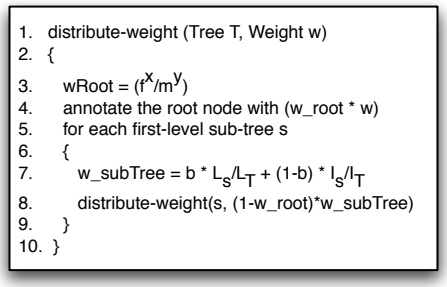

Figure 3. The *distribute-weight* algorithm

5 Deep-similar

Tree node weighing consists in associating a weight value —belonging to the interval [0,1]— to the nodes of an XML tree, depending on their position within the tree. Unlike other weighing techniques [29] the weighing algorithm we propose is designed to maintain a fundamental invariant: the sum of the weights associated to tree nodes must be equal to 1. Weighting techniques based on distance from root and fan-out have been used in both semi-structured and object-oriented databases to assess data item relevance. However, the ”right” weighting to use may be application and even data-set dependent. Nevertheless, it can be compared and adjusted with respect to users perceptions, e.g., by having the user to provide weights for the schema items [30].
In our framework, this is ensured via a function called *distribute-weight*. As seen in Figure 3, *distribute-weight* is a recursive function that traverses the XML tree, annotating each node with the appropriate weight. The weighting algorithm is initially called passing it the entire tree to be weighed. The first step is to decide how much of the total available weight (1) should be associated with the root-node (code line 3). Intuitively, the weight —and therefore the importance— of any given node must be directly proportional to the number of first-level children nodes it possesses (variable *f* in code line 3) and inversely proportional to the total number of nodes in the tree (variable *m* in code line 3). It is reasonable to assume that this relationship is probably document-set specific, as it quantitatively represents the ratio of document semantics carried by the root with respect to the one of internal elements; so it cannot be fixed once and for all. In our approach, it is calibrated through the use of two parameters, *x* and *y*, that are dataset dependant. In our scenario, after a number of experiments, parameter values 0.2246 and 0.7369 (respectively) have proven to give good results. In the example shown in Figure 4, we are interested in finding LOMs which include similar educational metadata.
The FuzzyXPath selection query below (see Section 7)

```
\LOM{[deep-similar(Educational,\LOM[1]\Educational] }
```

asks the system to provide just that. The sub-tree rooted in `Educational` from the left-hand side LOM is compared for similarity to the sub-tree rooted in `Educational` from the right-hand side LOM. Therefore, the algorithm is initially called passing to it as parameters the sub-tree starting at node `Educational` and a weight of 1 to be

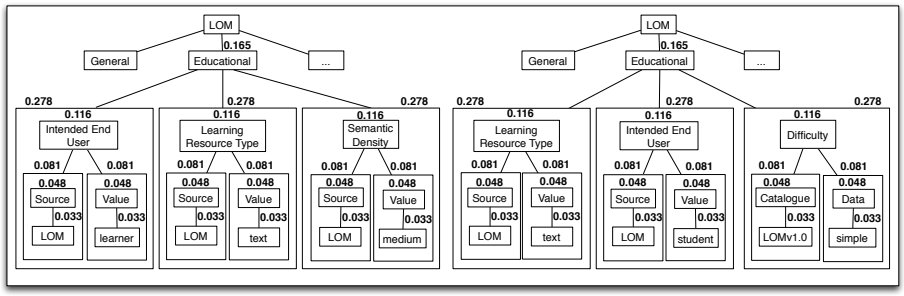

Figure 4. The example

distributed. The variable *wRoot* is calculated to be $3^{0.2246}/16^{0.7369}$, which is equal to 0.165. The second step consists in deciding how to distribute the remaining weight onto the sub-trees. This cannot be done by simply looking at the number of nodes each of these sub-trees possesses. Intuitively, the importance —and therefore the weight— of a sub-tree depends both on the amount of data it contains (stored typically in its leaf nodes) and on the amount of semantics encoded in its structure (typically, represented by its intermediate nodes). These considerations are taken into account in code line 7, in which variable L_s indicates the number of leaf nodes in sub-tree *s*, while the variable L_T indicates the number of leaf nodes in tree *T*. On the other hand, variable I_s indicates the number of intermediate nodes in sub-tree *s*, while variable I_T indicates the number of intermediate nodes in tree *T*. Again, we need to tune our weighting to the specific scenario by setting a balance between the semantics carried by leaves and by intermediate nodes. This balance is set by tweaking parameter *b*. Our experiments show that in our scenario our algorithm is more effective if we balance the formula slightly in favor of the leaf nodes by setting *b* to 0.6. Code line 7, however, only gives the percentage of the remaining weight that should be associated to each sub-tree. The actual weight is calculated in code line 8 through an appropriate multiplication, and passed recursively to the *distribute-weight* together with the respective sub-tree.
In our running example, the remaining weight $(1 - 0.165)$, equal to 0.835, is distributed onto the three sub-trees. The percentage of the weight that is associated to sub-tree *Intended End User* is $0.6 * 2/6 + 0.4 * 2/6$, which is equal to 0.333 (see code line 7). Therefore, the algorithm is called recursively passing sub-tree *Intended End User* and a weight of $0.835 * 0.333$, which is equal to 0.278. For the sake of conciseness, the weights associated to the remaining nodes are shown directly in Figure 4.

5.1 Tree Node Matching

Tree Node Matching is the last step in determining which Tree Edit Operations must be performed. Its goal is to establish a matching between the nodes in the source and the destination trees. Whenever a match for a given a node *n* cannot be found in the other tree, it is matched to the *null* value.
In this step, we use an algorithm that takes into account more complex structural properties the nodes might have, and —for the first time— semantic similarity. It scores

candidate nodes by analyzing how "well" they match the *reference node*. The candidate node with the highest score is considered the reference node's match.
Regarding structural properties, the algorithm considers the following characteristics: *number of direct children, number of nodes in the sub-tree, depth of the sub-tree, distance from the expected position, position of the node with respect to its father, positions of the nodes in the sub-tree, same value for the first child node*, and *same value for the last node*. Each of these characteristics can give from a minimum of 1 point to a maximum of 5 points.
Regarding semantic similarity, the algorithm looks at the nodes' tag names. These are confronted using, once again, Wordnet's system of hyperonyms (see Section 4.1). If the two terms are exactly the same, 1 point is given, if not their degree of similarity —a value in the set [0,1]— is considered.

5.2 Costs

Techniques presented in the above Sections 5 and 4.3 determine the costs the Tree Edit Operations have in our framework:

- The cost of the *Insert* operation corresponds to the weight the node being inserted has in the *destination* tree.
- The cost of the *Delete* operation corresponds to the weight of node being deleted from the *source* tree.
- The cost of the *Modify* operation can be seen as the deletion of a node from the *source* tree, and its subsequent substitution by means of an insertion of a new node containing the new value. This operation does not modify the tree's structure, it only modifies its content. This is why it is necessary to consider the degree of similarity existing between the node's old term and its new one. The cost is therefore $k * w(n) * (1 - Sim(n, destinationNode))$, where *w(n)* is the weight the node being modified has in the *source* tree, the function *Sim* gives the degree of similarity between node *n* and the destination value, and *k* is a tuning parameter dataset dependant. Experience in our scenario has shown that a good value for *k* is (0.9).
- The *Permute* edit operation does not modify the tree's structure. It only modifies the semantics (if any) that is implicit in the nodes' order. Therefore, its cost is $h * [w(a) + w(b)]$, where *w(a)* is the weight of node *a*, *w(b)* is the weight of node *b*, and *h* is a tuning parameter. Again, experience has shown that a good value for *h* is (0.36).

All the above cost-formulas are directly proportional to the weights of the nodes being treated. All operations, except *Permute*, can also be used directly on sub-trees as long as their total weights are used.
In the case of our running example (see Figure 4), the matchings, the corresponding tree edit operations, and their costs are presented in the following table. The cost of the *Modify* operation in the third row is computed as $0.9 * 0.033 * (1 - 0.769)$, while the cost of the *Permute* operation in row four is computed as $0.36 * [0.278 + 0.278]$.

Source	*Destination*	*Operation*	*Cost*
null	Difficulty	Insert	0.278
Semantic Density	null	Delete	0.278
learner	student	Modify	0.006
Intended End User / Learning Resource Type	NA	Permute	0.2

The degree of similarity between the two `Educational` sub-trees can therefore be calculated as $0.5^{(0.278+0.278+0.006+0.2)}$ (in this example we suppose $k = 0.5$ in the similarity formula), which evaluates to 0.589. This means that the `Educational` sub-trees of the two LOMs are quite different. If we had applied the same algorithm to the entire LOM trees, had they differed only in their `Educational` sub-trees, a much higher degree of similarity would have been obtained.

6 Integrated Query Evaluation

We are now ready to show integrated execution of structure- and content-related flexible queries. Fuzzy constraints on content and structure can be composed within a single query by means of simple logical connectives, that, in this context, are redefined to take into account the different semantics of the flexible constraints. Following the standard fuzzy approach [31], extended logical connectives can be straightforwardly defined as follows:

- `NOT` The negation preserves the meaning of complement, considering a fuzzy set A, with membership function μ, for each element a, if $a \in A$ has membership $\mu(a)$ then $a \in \bar{A}$ has membership $1 - \mu(a)$.
- `AND` Given two fuzzy sets A_1, A_2 with membership functions μ_1, μ_2, and given the elements $a_1 \in A_1, a_2 \in A_2$ then $a_1\ AND\ a_2$ is traditionally equal to the intersection of the two elements that in the Fuzzy Set Theory is evaluated by means of a T-norm function as, for instance, the min function. Hence $\mu(a_1\ AND\ a_2) = min(\mu_1(a_1), \mu_2(a_2))$.
- `OR` Given two fuzzy sets A_1, A_2 with membership functions μ_1, μ_2, and given the elements $a_1 \in A_1, a_2 \in A_2$ then $a_1\ OR\ a_2$ is evaluated by means of a T-conorm function as, for example, the max function. Hence $\mu(a_1 OR a_2) = max(\mu_1(a_1), \mu_2(a_2))$.

As one would expect, the interpretation of the disjunction is symmetric with respect to conjunction. Consider now a query combining structure and content-related flexible features:

```
\LOM{[deep-similar(educational,\LOM[1]\educational] AND
                                        \\duration CLOSE PT1H }
```

The evaluation of such a query is the result of a four-steps process:

1. The query is transformed into a crisp one, capable of extracting data guaranteed to be a superset of the desired result. In the example we obtain `LOM` which extracts all the LOMs, which are clearly a superset of the desired result.
2. Each fuzzy constraint p_i is evaluated w.r.t. each of the extracted data's items, and a degree of satisfaction v_i is assigned to it.
   In this example, we evaluate deep-similarity between `Educational` sub-items, and then take into account to what degree the LOM's duration is `CLOSE` to one hour (e.g., using a dictionary). For the first item of the result-set (the right-hand side `LOM` in Figure 4), the former evaluates to 0.238, and the latter to 0.67 (45 minutes against one hour).
3. An overall degree of satisfaction is obtained for each item in the result. This is done considering all constraints of the path expression in conjunction. In our example, we take the smallest of the two degrees of satisfaction (0.238).
4. The items in the result are ordered according to the degree of satisfaction.

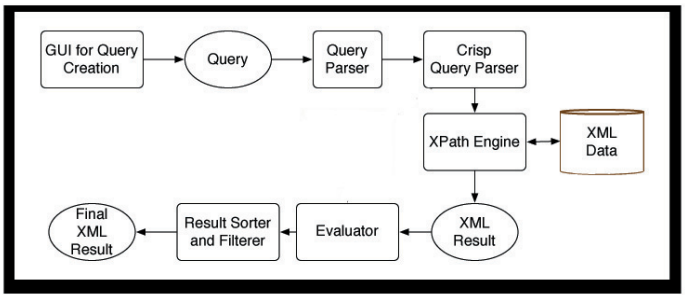

Figure 5. FuzzyXPath client architecture.

6.1 User-centered Query Execution

Even if deep-similarity greatly increases FuzzyXPath recall w.r.t. standard XPath, for some advanced applications even the idea of taking all flexible constraints in conjunction can be too rigid. An alternative, a user-centered approach can be taken allowing advanced users to explicitly bind degrees to variables and to define their own function to calculate the final degree of satisfaction. To this end, we define a `WITH RANKING` clause in order to combine the bound values in the final ranking. The following example shows a FuzzyXPath expression with two fuzzy conditions:

```
\LOM{[deep-similar(educational,\LOM[1]\educational] | v1 AND
              \\duration NEAR PT1H | v2 WITH RANKING v1 * v2}
```

The ranking of the result set is obtained as the product of the values bound to $v1$ and $v2$. More complex `WITH RANKING` clauses can involve linear combination of the values bound to the ranking variables (e.g., `WITH RANKING 0.4*v1 + 0.6*v2`) as well as a normalized weighted average.

7 Implementation of FuzzyXPath Query Execution Engine

In this Section, we discuss some issues regarding FuzzyXPath implementation. Figure 6 illustrates the prototype software environment we have developed (using .NET 2.0 technology) for the execution of FuzzyXPath queries. The tool allows us to import any valid XML data file, to display its contents in a special-purpose text pane, and to perform FuzzyXPath queries against it. Queries can be typed manually, or loaded from a file in which they have been previously stored, and remain visible at all times on the left hand side of the interface. The results for executed queries can be seen in the `XML Result` pane, or in a special "ranked" format in the `Ranked Result` pane. The result cardinality (i.e., the number of returned nodes) is show at the bottom of the interface, together with the total query execution time. Query results can also be filtered by indicating the maximum ranking value to be accepted. Finally, our tool also supports the definition of "projects", in which we jointly store an XML data source and a set of FuzzyXPath queries.

The *interface* (see Figure 6) is mainly devoted to editing the queries. The user is assisted with a strong syntactic feedback that prevents the composition of incorrect queries.

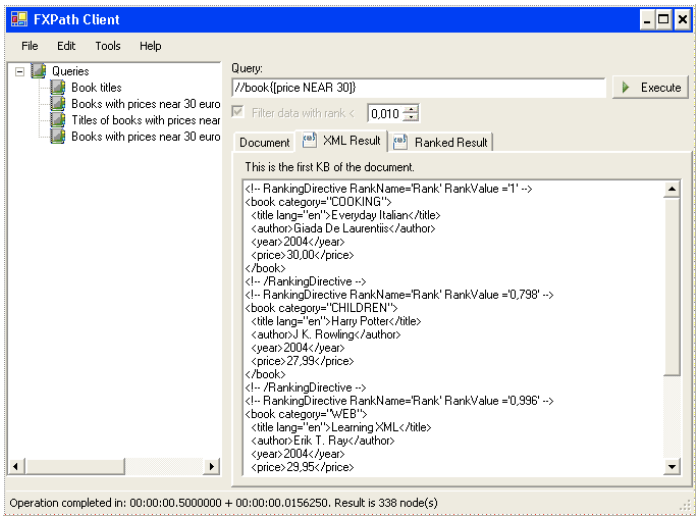

Figure 6. FuzzyXPath client interface.

When a query has been completed, it is sent to the *engine*, where it goes through a sequence of steps before the engine returns the result set. First of all, the query is parsed and translated into a set of crisp queries that can be managed by a standard XPath engine. The nature of the translation depends on the fuzzy constraints used in the original query.

The results of the crisp queries are then passed to a special-purpose fuzzy constraint evaluator. Once the fuzzy constraints have been evaluated, the results are sorted and filtered to produce the end results.

Our current FuzzyXPath query execution environment consists of about 80 C# classes. An overall picture of the system's architecture is shown in Figure 5.

7.1 Evaluation

In order to provide a preliminary evaluation of the performance of our implementation, we defined a set of three simple queries and executed them against different sized data files. The data files used for the evaluation contain references to publications, used as teaching material, much in the line of our sample scenario.

The first query (referenced as *Single*, since it only uses one fuzzy constraint) asks the engine to return all the publications that have a price that is `close` to thirty euros. Such a query is written in FuzzyXPath as

`//publication{[price CLOSE 30]}`.

The second query (referenced as *Two-step*) asks the engine to return the titles of the publications that have a price that is `close` to thirty euros. This query is written in FuzzyXPath as

`//publication{[price CLOSE 30]}/title`.

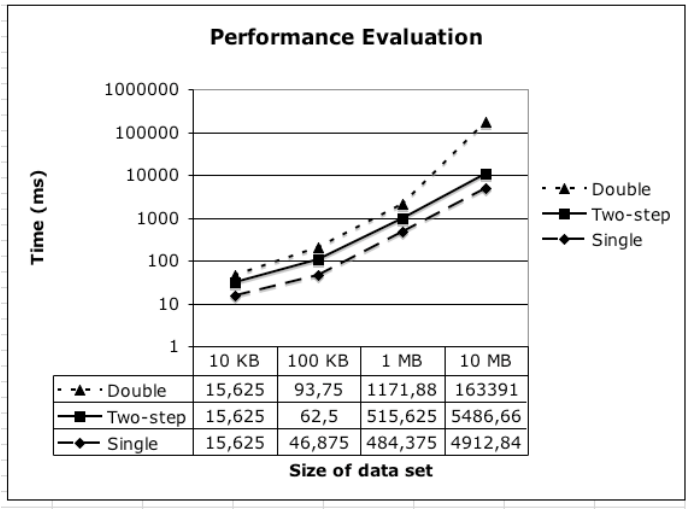

Figure 7. Query performance

The third query (referenced as *Double*, since it introduces a second fuzzy constraint) asks the engine to return all publications whose price that is `near` thirty euros and that were published in a year `close` to 2003. This query is written in FuzzyXPath as `//publication{[price CLOSE 30 and year CLOSE 2003]}`.

These three queries were executed against four differently sized data sets. The first was circa $10KB$ in size and contained 56 publication references; the second was circa $100KB$ in size and contained 596 publication references; the third was circa $1MB$ in size and contained 6052 publication references; the fourth was circa $10MB$ in size and contained 60972 book references. The performance evaluation tests were performed on an AMD Athlon XP $2600 + 1.92$ GHz computer, with 512MB of RAM running Windows XP Service Pack 2.

Figure 7 presents a stacked-line chart of the query results. As we can see, the tool performs quasi-logarithmically. On the x-axis we have each step represents a tenfold increase in the size of the data set being used. Similarly, on the y-axis we have a tenfold increase of the time. We experienced a deviation from the expected behavior when performing the *Double* query on the 10MB data set. The reason is due to the saturation of our CPU.

Table 1, on the other hand, gives the number of nodes returned by each query for each data set.

Similar experiments were performed using other flexible constraints, obtaining results that confirmed that the tool performance are good. As an example, we evaluated the query `deep-similar(/LOM/book, /LOM/book[1])` obtaining the stacked-line chart of the query results shown in Figure 8. In this case the graphic shows a linear time increase with respect to the size — number of nodes — of the documents. Note that if the complexity of the nodes to compare increases, the evaluation time in-

Size of data set	Single	Two-step	Double
10 KB	32	35	23
100 KB	338	371	240
1 MB	3444	3767	2476
10 MB	34842	38108	25042

Table 1. Number of nodes returned by the queries.

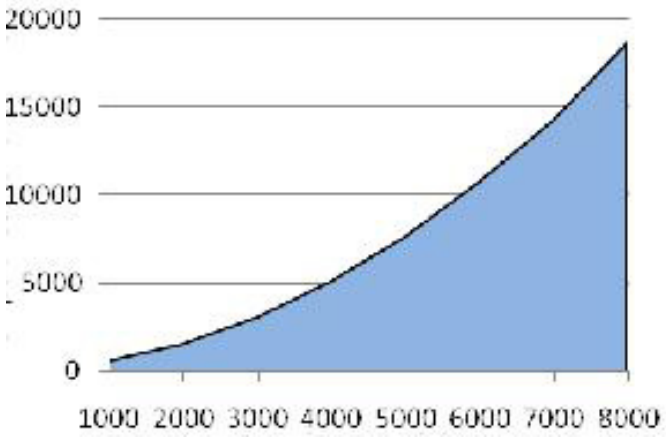

Figure 8. Query performance with the deep-similar constraint, the x-axis shows the number of document nodes while the y-axis represents the execution time in ms

creases exponentially with respect to this complexity (that is in general limited in real use cases).

8 Conclusion and Outlook

In this paper, we presented FuzzyXPath, a complete framework for querying semi-structured XML data based on Fuzzy Set Theory. FuzzyXPaths main advantage is preventing silence that can be caused by (1) data not following a schema precisely, (2) the user providing a blind query (e.g., because she does not know the schema or exactly what she is looking for), and (3) data being produced using slightly diverse schemas, e.g., due to schema evolution). All three phenomena listed above are highly relevant for practical applications, specifically, they have been documented where users of XML query environments were provided with schema summaries rather than with complete schemas. FuzzyXPath flexibility is achieved through the integrated evaluation of fuzzy constraints, and fuzzy tree matching. Both evaluations must rely on domain semantics for dealing with variations in vocabulary: our approach currently relies on external dictionaries like WordNet for calculating semantic similarity. However, we believe that more small domain specific dictionaries should be used to obtain better results and to improve performance at the same time. Our future work will, in fact, concentrate on further validating our approach using domain specific dictionaries.

References

1. Yu, C., Jagadish, H.V.: Xml schema summarization. (In: VLDB 2006) 319–330
2. Braga, D., Campi, A., Damiani, E., Pasi, G., Lanzi, P.L.: FXPath: Flexible querying of xml documents. In: Proc. of EuroFuse. (2002)
3. Mouchaweh, M.S.: Diagnosis in real time for evolutionary processes in using pattern recognition and possibility theory (invited paper). International Journal of Computational Cognition **2**(1) (2004) 79–112 ISSN 1542-5908.
4. Loiseau, Y., Prade, H., Boughanem, M.: Qualitative pattern matching with linguistic terms. AI Commun. **17**(1) (2004) 25–34
5. Kacprzyk, J., Ziolkowski, A.: Database queries with fuzzy linguistic quantifiers. IEEE Trans. Syst. Man Cybern. **16**(3) (1986) 474–479
6. Bosc, P., Pivert, O.: Fuzzy querying in conventional databases. (1992) 645–671
7. Bosc, P., Lietard, L., Pivert, O.: Soft querying, a new feature for database management systems. In: DEXA. (1994) 631–640
8. Bosc, P., Lietard, L., Pivert, O.: Quantified statements in a flexible relational query language. In: SAC '95: Proceedings of the 1995 ACM symposium on Applied computing, New York, NY, USA, ACM Press (1995) 488–492
9. Galindo, J., Medina, J., Pons, O., Cubero, J.: A server for fuzzy sql queries. In: Proceedings of the Flexible Query Answering Systems. (1998)
10. Dubois, D., Prade, H., Sedes, F.: Fuzzy logic techniques in multimedia database querying: A preliminary investigation of the potentials. IEEE Transactions on Knowledge and Data Engineering **13**(3) (2001) 383–392
11. Damiani, E., Tanca, L.: Blind queries to xml data. In: DEXA LNCS 1873. (2000) 345–356
12. Damiani, E., Oliboni, B., Tanca, L.: Fuzzy techniques for XML data smushing. In: Proceedings of the International Conference, 7th Fuzzy Days on Computational Intelligence, Theory and Applications, London, UK, Springer-Verlag (2001) 637–652
13. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree pattern relaxation. In Springer, ed.: Proceedings of EDBT. (2002)
14. Schlieder, T.: Schema-driven evaluation of approximate tree-pattern queries. In Springer, ed.: Proceedings of EDBT. (2002)
15. Li, H.G., Aghili, S.A., Agrawal, D., Abbadi, A.E.: FLUX: Fuzzy content and structure matching of XML range queries. In: Proceedings of WWW 2006, May 23-26, 2006, Edinburgh, Scotland. (2006)
16. Ciaccia, P., Penzo, W.: The *collection index* to support complex approximate queries. In Verlag, S., ed.: Proceedings of XSym 2003. (Volume 2824.) 164–179
17. Buche, P., Dibie-Barthèlemy, J., Wattez, F.: Approximate querying of XML fuzzy data. In Springer, ed.: Proceedings of the 7th International Conference FQAS 2006, Milan, Italy. (2006)
18. Mandreoli, F., Martoglia, R., Tiberio, P.: Approximate query answering for a heterogeneous XML document base. In Springer, ed.: Proceedings of the 5th Int. Conf on Web Information Systems Engineering, Brisbane, Australia, November 22-24 (2004)
19. Amer-Yahia, S., Lakshmanan, L.V.S., Pandit, S.: Flexpath: flexible structure and full-text querying for xml. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2004) 83–94
20. W3C: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath` (1999)
21. Zadeh, L.: Fuzzy sets. Information and Control. **8**(4) (1965) 338–353
22. Damiani, E., Lavarini, N., Oliboni, B., Tanca, L.: An approximate query environment for xml data. In V. Loia, M. Nikravesh, L. Zadeh (Eds.), Fuzzy Logic and the Internet,Studies in Fuzziness and Soft Computing **137** (2004) 71–94

23. Simon, D.: Sum normal optimization of fuzzy membership functions. Int. J. Uncertain. Fuzziness Knowl.-Based Syst. **10**(4) (2002) 363–384
24. Patrick Bosc, E.D., Fugini, M.: Fuzzy service selection in a distributed object-oriented environment. IEEE Transactions on Fuzzy Systems **9**(5) (2001) 682–698
25. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics-Doklady **10** (1966) 707–710
26. Nierman, A., Jagadish, H.V.: Evaluating structural similarity in XML documents. In: Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002), Madison, Wisconsin, USA (2002)
27. Garofalakis, M., Kumar, A.: Correlating xml data streams using tree-edit distance embeddings. In: PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, ACM (2003) 143–154
28. Reis, D.C., Golgher, P.B., Silva, A.S., Laender, A.F.: Automatic web news extraction using tree edit distance. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, New York, NY, USA, ACM (2004) 502–511
29. Damiani, E., Tanca, L., Arcelli-Fontana, F.: Fuzzy xml queries via context-based choice of aggregations. Kybernetika **16**(3) (2000)
30. Ceravolo, P., Damiani, E., Viviani:, M.: Bottom-up extraction and trust-based refinement of ontology metadata. IEEE Trans. Knowl. Data Eng. **19**(2, pages = 149-163) (2007)
31. George J. Klir and Bo Yuan: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice-Hall (1995)