An Operational Semantics for XML Fuzzy Queries

Alessandro Campi[1], Sam Guinea[1] and Paola Spoletini[2]
[1]Politecnico di Milano, DEIB, p.zza Leonardo da Vinci 32, 20133, Milano, Italy
[2]Universit dell'Insubria, DSTA, Via Mazzini 5, 21100 Varese, Italy

Keywords: Fuzzy, XML.

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 define binary predicates. In this paper, we advocate that binary selection can at times be restrictive due to very nature of XML, and to the uses that are made of it. We therefore suggest a querying framework, called FXPath, based on fuzzy logics. In particular, we propose the use of fuzzy predicates for the definition of "vaguer" and "softer" queries. We also introduce a function called "deep-similar", which aims at substituting XPath's typical "deep-equal". Its goal is to provide a degree of similarity between two XML trees, assessing whether they are similar both structure-wise and content-wise. In this paper we present the formal syntax and semantics of FXPath, and discuss implementation issues.

1 INTRODUCTION

The principal proposals for querying XML documents have been XPath and XQuery. Both XPath and XQuery divide data into those which fully satisfy the selection conditions, and those which do not. However, binary conditions can be —in some scenarios— a limited approach to effective querying of XML data. The reasons behind such a statement lie in the very nature of XML. Even though a standard for defining how data must be structured within an XML file exists, namely the XML Schema, it is often the case that end users have to work with data that does not have a schema, or for which a schema is not known at the moment of the query.

A few considerations can be made to justify this claim. First of all, even when XML schemas do exist, data producers do not always follow them precisely. Second, users often end up defining *blind* queries, either because they do not know the XML schema in detail, or because they do not know exactly what they are looking for.

In this paper, we give a formal semantics and an implementation to a framework (Campi et al., 2006; Braga et al., 2002) for querying XML data that goes beyond binary selection, and that allows the user to define "vaguer" and "softer" selection conditions. The main idea is that the constraint evaluation produces a fuzzy subset, and associates a numeric value to each information item (i.e., the membership degree). The extensions introduced to XPath can be divided into three main strains: fuzzy axis navigation, fuzzy predicate evaluation, and the fuzzy function "deep-similar".

In this paper we use a simple library-based case study for the evaluation of FXPath shown in Figure 1. For example, a simple fuzzy query on such a data set could be used to find all the books that were published near to the year 2000. To do so we state a predicate on a book's "year" attribute. Since the attribute is a number we use the fuzzy comparator "=". We might retrieve books published in the year 2000, 2001, 2002, etc.

```
<!--RankingDirective RankingValue="1.0"-->
 <Book year="2000"><title>t1</title>
 </Book>
<!-- /RankingDirective -->
<!-- RankingDirective RankingValue=".8"-->
 <Book year="2001"><title>t2</title>
 </Book>
<!-- /RankingDirective -->
```

Notice the presence of a ranking directive inserted as a comment that contains each result's ranking value, i.e., a value from the set $[0,1]$. The closer it is to 1, the better the returned item satisfies the condition (i.e. having been published in a year near the year 2000).

Campi A., Guinea S. and Spoletini P..
An Operational Semantics for XML Fuzzy Queries.
DOI: 10.5220/0005155502050210
In *Proceedings of the International Conference on Fuzzy Computation Theory and Applications* (FCTA-2014), pages 205-210
ISBN: 978-989-758-053-6

```
<bib> <!-- Tree K -->
  <book year="2001"> <!-- K1 -->
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price> 65.95</price>
    <section id="intro" difficulty="easy" > <!-- S1 -->
     <title>Introduction</title>
     <p>Text ... </p>
     <section>  <title>Audience</title> <!-- S2 -->
          <p>Text ... </p>  </section>
      <section> <!-- S3 -->
       <title>Web Data and the Two Cultures</title>
       <p>Text ... </p>
       <figure height="400" width="400">
         <title>Client/server architecture</title>
       </figure>
      <p>Text ... </p>
      </section>
  </book>
  ...
</bib>
```

Figure 1: Running example.

2 RELATED WORK

Fuzzy sets have been shown to be a convenient way to model flexible queries in (Bosc et al., 1994). Many attempts to extend SQL with fuzzy capabilities have been undertaken in recent years. (Bosc et al., 1995) describes SQLf, a language that extends SQL by introducing fuzzy predicates that are processed on crisp information.

Several approaches have been proposed to introduce flexibility in semi-structured information processing. An early technique (Damiani and Tanca, 2000) was based on fuzzy encoding of XML data trees. A later paper (Amer-Yahia et al., 2002) proposed an approach based on XML query rewriting, supporting renaming and deletion of nodes in the query. Hybrid techniques (Schlieder, 2002) have also been proposed, where XML data are encoded and queries are rewritten. A recent approach to this problem (Li et al., 2006) proposes a dynamic summarization and indexing method called FLUX.

In (Bosc et al., 2006) the authors propose to relax failing queries, based on a notion of proximity. In (Sanz et al., 2006) the authors tackle the problem of highly heterogenous XML collections, in which data pertaining to a certain domain are collected within documents that are highly diverse in structure. In (Sanz et al., 2008) the authors refine these concepts and formally define them, they improve the algorithms used for the matching, and analyze their complexity.

Finally, we illustrate two cases in which fuzzy information retrieval is used within specific domains. In (Bordogna et al., 2006) the authors present a news filtering model based on fuzzy hierarchical categorization. In (Bandini et al., 2006) the authors describe the architecture of a query answering system for the domain of motor racing using fuzzy logic and domain-specific knowledge.

(Amer-Yahia et al., 2004) 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. Recently, in (HadjAli and Pivert, 2008), a fuzzy technique to reduce the number of views that are satisfactory w.r.t. the query is present.

3 FXPATH

3.1 Formal Semantics

The XML Path Language (XPath) uses a declarative notation: each expression developed from this notation describes the types of nodes that need to be matched, based on the hierarchical relationships existing between the nodes. We propose to extend the XPath language definition with constructs for the specification of fuzzy predicates and fuzzy subsets. The effect is an increase and improvement of the results produced by the query evaluation. Figure 2 shows the syntax extensions over XPath.

The operational semantics of FXPATH, that can be use straightforward to evaluate a query, are defined over a "forest" $F = \{T_1, \dots, T_{|F|}\}$ of couples

$$T_i = (XMLTree, evaluation).$$

Given a couple T (or a sequence of couples), we use function $tree(T_1, ..., T_n)$ to obtain its $XMLTree$ (or the sequence of $XMLTree$s), and function $value(T_1, ..., T_n)$ to obtain an $evaluation$ (or the sequence of $evaluation$s), i.e., a value (or a sequence of values) in the set $[0,1]$. The execution of a generic query q on a forest F results in the union of the execution of the query on all the "trees" in F. It can be expressed as: $eval(q,F) = order\_cut(\bigcup_{i=1}^{|F|} eval(q,T_i))$ where $order\_cut$ sorts the set of execution results, and eliminates those that do not reach a certain threshold.

FXPath queries are executed recursively, in accordance with the intrinsic recursiveness of their structure. The entrance point is function $eval\_query$, which takes a syntactically correct FXPATH query and a couple $(XMLTree, evaluation)$, and returns a (possibly empty) sequence of couples $(XMLTree, evaluation)$.

```
[1]  LocationPath ::= RelativePath | AbsolutePath
[2]  AbsolutePath ::= '/' RelativePath? | '//' RelativePath
[3]  RelativePath ::= Step | Step '/' RelativePath | Step '//' RelativePath
[4]  Step ::= AxisSpecifier NodeTest ( '[' Predicate ']' )? | '.' | '..'
[5]  AxisSpecifier ::= AxisName '::' | '@' | '{' AxisName '::' '}' | '{@}' |
[6]  AxisName ::= 'ancestor' | 'ancestor-or-self' | 'attribute' | 'child' | 'descendant' | 'descendant-or-self' |
    'following' | 'following-sibling' | 'namespace' | 'parent' | 'preceding' | 'preceding-sibling' | 'self'
[9] Predicate ::= Predicate 'and' Predicate | Predicate 'or' Predicate | '('Predicate')' | 'not(' Predicate ')' | CompExpr
[9bis] Predicate ::= '{' CompExpr '}'
[10] CompExpr ::= Expr ( '=' | '!=' | '<' | '>' | '<=' | '>=' ) Expr | Expr
[14] Expr ::= ('-')? Expr | Expr ( '+' | '-' | '*' | 'div' | 'mod' ) Expr | '(' Expr ')'
    | PrimaryExpr | LocationPath | Expr 'near' Expr | '{' Expr '}'
[15] PrimaryExpr ::= VariableReference | Literal | Number | FunctionCall
[16] FunctionCall ::= FunctionName '(' ( Expr ( ',' Expr )* )? ')'
[35] FunctionName ::= Function names defined in XPath
[35bis] FunctionName ::= 'deep-similar'
[7]  NodeTest ::= NameTest | NodeType '(' ')' | 'processing-instruction' '(' Literal ')'
[29] Literal ::= '"' [^"]* '"' | "'" [^']* "'"
[30] Number ::= ( [0-9]* ('.' [0-9]*)? )
```


Figure 2: EBNF specification of Fuzzy XPath.

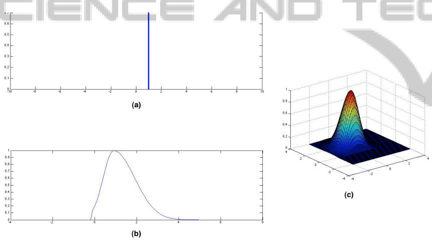

Figure 3: Crisp and Fuzzy Membership Functions.

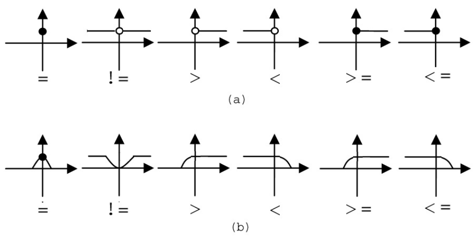

Figure 4: Crisp and Fuzzy Comparators.

The XML data on which the query is evaluated are seen as global variables and the tree component of the parameter T is the pointer to one of the nodes of the XML tree. Notice that if the cardinality of the sequence is greater than one, function *order_cut* is used.

Now we describe the evaluation of a generic query. By definition an *AbsolutePath* is either $/RelativePath$ or $//RelativePath$. Therefore,

$$eval\_query(AbsolutePath, T) = eval\_query(/RelativePath, T)$$

or

$$eval\_query(AbsolutePath, T) = eval\_query(//RelativePath, T)$$

These two evaluations, on the other hand, are equivalent to the evaluation of the query's components starting from the root node of the tree pointed by T (given by $root(tree(T))$), i.e.,

$$eval\_query(/RelativePath, T) = eval\_query\_components(/RelativePath, (root(tree(T)), value(T)))$$

and

$$eval\_query(//RelativePath, T) = eval\_query\_components(//RelativePath, (root(tree(T)), value(T)))$$

and

$$eval\_query(RelativePath, T) = eval\_query\_components(/RelativePath, T)$$

Function *eval_query_components* takes as input a query and a couple $(XMLTree, evaluation)$, and returns zero or more couples $(XMLTree, evaluation)$. Once again, if a sequence is returned, the behavior is as in the case of *eval_query* (i.e, function *order_cut* is used). Query components are evaluated recursively, and the recursive step depends on the structure of the query. Our base case is the empty query, for which

$$eval\_query\_components(, T) = T$$

The evaluation of a query structured as $/RelativePath$ (or $//RelativePath$) can be substituted with the evaluation of $/stepAbsolutePath$ (or $//stepAbsolutePath$). This is a direct consequence of how the syntax is defined. Therefore

$$eval\_query\_components(/RelativePath, T) = eval\_query\_components(/stepAbsolutePath, T)$$

For the same reason it is also true that

$$eval\_query\_components(RelativePath, T) = eval\_query\_components(stepAbsolutePath, T)$$

Likewise, since an *AbsolutePath* is either a */RelativePath* (or a *//RelativePath*), the substitution can also take place the other way around. Therefore, it is also true that

$$eval\_query\_components(AbsolutePath, T) = eval\_query\_components(/RelativePath, T)$$

At this point the evaluation of a query of the form

$$stepAbsolutePath$$

can be achieved evaluating a query of the form

$$AxisSpecNodeText[Pred]AbsolutePath$$

This is due to the fact that a single XPath step is made up of a axis specification, a node text, and a predicate. The evaluation of such a query is achieved by using the axis specification and the node text to select a set of nodes from the current tree, and then applying the predicate to this set. This is why

$$eval\_query\_components (AxisSpecNodeText[Pred]AbsolutePath, T) = eval\_query\_components([Pred]AbsolutePath, eval\_on\_tree(/AxisSpecNodeText, T))$$

where function *eval_on_tree* takes a navigation instruction and performs it on a tree. Similarly,

$$eval\_query\_components (/AxisSpecNodeText[Pred]AbsolutePath, T) = eval\_query\_components([Pred]AbsolutePath, eval\_on\_tree(/AxisSpecNodeText, T))$$

and

$$eval\_query\_components (//AxisSpecNodeText[Pred]AbsolutePath, T) = eval\_query\_components([Pred]AbsolutePath, eval\_on\_tree(//AxisSpecNodeText, T))$$

More details regarding function *eval_on_tree* will be given shortly. In the meanwhile, once the navigation step has been completed, the predicate is evaluated, and after this evaluation the next step in the query is taken. Indeed,

$$eval\_query\_components([Pred]AbsolutePath, T) = eval\_query\_components(AbsolutePath, eval\_query\_components(Pred, T))$$

The evaluation of the predicate on a tree returns zero or more of couples $(XMLTree, evaluation)$, and is given by

$$eval\_query\_components(pred, T) = (tree(T), min(value(T), eval\_predicate(pred, tree(T))))$$

It is calculated as the minimum between the value associated with the tree and the evaluation of the predicate on that tree, as given by function *eval_predicate*.

The three functions

$$eval\_on\_tree(AxisSpecNodeText, T)$$

$$eval\_on\_tree(/AxisSpecNodeText, T)$$

$$eval\_on\_tree(//AxisSpecNodeText, T)$$

use the appropriate crisp navigation functions to choose a finite set of nodes from the current tree. Indeed, it is not limited to nodes that satisfy the axis relationship in a crisp sense, but extends the set with nodes that satisfy the relationship in a "fuzzy" sense.

The function *eval* returns the evaluation (i.e., a value in the set $[0,1]$) of a predicate on a given tree K. The evaluation of the disjunction (*or*) of two predicates is the highest evaluation amongst the two, the evaluation of the conjunction (*and*) of two predicates corresponds to the minimum evaluation amongst the two, and negation follows the typical fuzzy definition. If the predicate is a path expression p of any kind (absolute or relative),

$$eval\_predicate(p, K) = max(value(eval\_query\_component(p, (K, 1))))$$

where function *max* returns the maximum evaluation. Regarding the evaluation of comparators, we distinguish between crisp version and fuzzy version. The crisp version returns 1 if the comparator is satisfied, and 0 if it is not. Fuzzy comparatorsreturn a value in the set $[0,1]$ depending on the comparator's membership function (defined by μ). Chosen an expression (e.g., $expr_2$), μ is calibrated using the expression's evaluation (in this case $eval\_expr(expr_2, K)$), and evaluated against the other expression (in this case $eval\_expr(expr_1, K)$). For example, if $eval\_expr(expr_2, K)$ returns a numerical value 3, and we are dealing with the fuzzy comparator $\{>\}$, membership function μ may be defined in order to return 1 if $eval\_expr(expr_1, K)$ is greater than 3, and a value in the set $[0,1]$ if it is lesser than 3, depending on how far $eval\_expr(expr_1, K)$ is from that value (see Figure 4).

Function *eval_expr* takes as input two parameters: an expression, as defined by the grammar and a tree K and returns a tree (more often just a leaf). The different cases are defined as follows:

$$eval\_expr(expr1\ op\ expr2), K) = eval\_expr(expr1, K) op\ eval\_expr(expr2, K)$$

$$eval\_expr((expr), K) = eval\_expr(expr, K)$$

$$eval\_expr(-expr, K) = -eval\_expr(expr, K)$$

$$eval\_expr(variableReference, K) = val(variableReference)$$

where the function *val* returns the content of its argument.

$$eval\_expr(Literal, K) = val(Literal)$$

$$eval\_expr(Number, K) = Number$$

$$eval\_expr(FunctionCall, K) =$$
$$eval\_expr(FunctionName(expr_1, ..., expr_n), K) =$$
$$function(eval\_expr(expr_1, K), \ldots)$$

where $function$ is the XPath function being called. If the expression is a path p of any kind (absolute or relative)

$$eval\_expr(path, K) =$$
$$tree(max\_value(eval\_query_c omponent(p, (K, 1))$$

where function max_value returns the couple

$$(XMLTree, evalution)$$

with the maximum *evaluation*.

Notice that when *pred* is a number, it must be treated differently. Indeed, its semantics requires that we select the $n-th$ child of the current node.

3.2 Function Deep-similar

In FXPath we have added a new function called $deep-similar$. It is a fuzzy version of the classical XPath function $deep-equals$. A formal definition follows.

Definition 1 (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 contained in the set* $[0,1]$*. This degree of similarity is given as 1 - (the cost of transforming* T_1 *into* T_2 *using Tree Edit Operations). Therefore, if two trees are completely different, their degree of similarity is* 0*; if they are exactly the same —both structure-wise and content-wise— their degree of similarity is* 1.

The tree operations that can be used to transform the tree are defined as follows:

Definition 2 (Insert)**.** *Given an XML tree T, an 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. The cost of the Insert edit operation corresponds to the weight that the node being inserted has in the destination tree.*

Definition 3 (Delete)**.** *Given an XML tree T, and a location loc (defined through a path expression that selects a single node n in T), Delete(T, loc) transforms T into a new tree* T' *in which node n is removed. The cost of the Delete edit operation corresponds to the weight of the node being deleted from the source tree.*

Definition 4 (Modify)**.** *Given an XML tree T, a location loc, and a new value v, Modify(t, loc, v) transforms T into a new tree* T' *in which the content of node n is replaced by v. The cost of the Modify edit 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 only modifies the node content, so it is necessary to consider the similarity existing between the node's old and new term. This is achieved using Wordnet's system of hypernymys. 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 constant (0.9).*

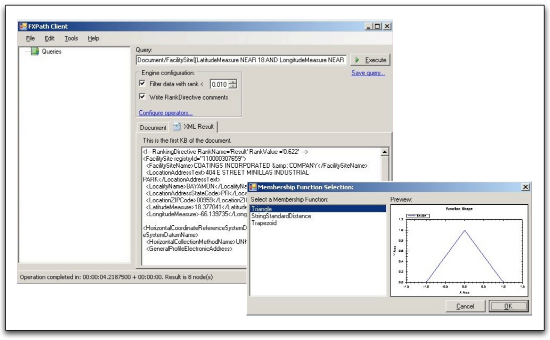

Figure 5: Fuzzy XPath client interface.

Definition 5 (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. The Permute edit operation does not modify the tree's structure. It only modifies the semantics that are intrinsically held in the order the nodes are placed in. Therefore, its cost is* $h * [w(a) + w(b)]$*, where w(...) is the weight of a node and h is a constant (0.36).*

4 TOOL

FXPath is fully implemented within a graphical environment shown in Figure 5 whose goal is to simplify the definition of queries, making the designer's job less error prone. Figure 6 shows how the execution proceeds. A query is sent to the *engine*, where it goes through multiple steps before returning 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. Since new fuzzy predicates can be added to the tool at any time, each predicate is associated with a set of rules for performing the

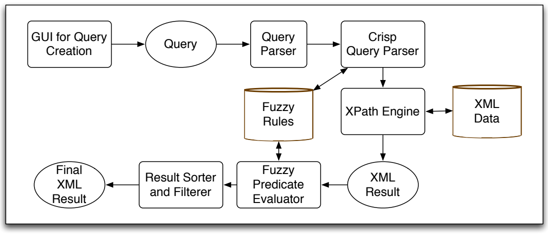

Figure 6: Fuzzy XPath client architecture.

translation. The results of the crisp queries are then passed to a fuzzy predicate evaluator. The information needed by this component is also contained in an extensible set of fuzzy rules. Once the fuzzy predicates have been evaluated, the results are sorted and filtered to produce the end results.

5 CONCLUSION

We have presented a framework for querying semi-structured XML data based on key aspects of fuzzy logics. Its main advantage is the minimization of the silent queries that can be caused by (1) data not following an appropriate schema faithfully, (2) the user providing a blind query in which he does not know the schema or exactly what he is looking for, and (3) data being presented with slightly diverse schemas. This is achieved through the use of fuzzy predicates, and fuzzy tree matching. In our future work, we are interested in extending the approach to the XQuery language, in order to achieve a fully-fledged fuzzy querying language for XML data sets.

REFERENCES

Amer-Yahia, S., Cho, S., and Srivastava, D. (2002). Tree pattern relaxation. In Springer, editor, *Proceedings of EDBT*.

Amer-Yahia, S., Lakshmanan, L. V. S., and Pandit, S. (2004). Flexpath: flexible structure and full-text querying for xml. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 83–94, New York, NY, USA. ACM Press.

Bandini, S., Mereghetti, P., and Radaelli, P. (2006). Fuzzy query answering in motor racing domain. In (Larsen et al., 2006), pages 295–306.

Bordogna, G., Pagani, M., Pasi, G., and Villa, R. (2006). A flexible news filtering model exploiting a hierarchical fuzzy categorization. In (Larsen et al., 2006), pages 170–184.

Bosc, P., HadjAli, A., and Pivert, O. (2006). Relaxation paradigm in a flexible querying context. In (Larsen et al., 2006), pages 39–50.

Bosc, P., Lietard, L., and Pivert, O. (1994). Soft querying, a new feature for database management systems. In *DEXA*, pages 631–640.

Bosc, P., Lietard, L., and Pivert, O. (1995). Quantified statements in a flexible relational query language. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 488–492, New York, NY, USA. ACM Press.

Braga, D., Campi, A., Damiani, E., Pasi, G., and Lanzi, P. L. (2002). FXPath: Flexible querying of xml documents. In *Proc. of EuroFuse*.

Campi, A., Guinea, S., and Spoletini, P. (2006). A fuzzy extension for the xpath query language. In *FQAS*, pages 210–221.

Damiani, E. and Tanca, L. (2000). Blind queries to xml data. In *DEXA LNCS 1873*, pages 345–356.

HadjAli, A. and Pivert, O. (2008). Towards fuzzy query answering using fuzzy views - a graded-subsumption-based approach. In *ISMIS*, pages 268–277.

Larsen, H. L., Pasi, G., Arroyo, D. O., Andreasen, T., and Christiansen, H., editors (2006). *FQAS 2006, Milan, Italy, June 7-10, 2006*, volume 4027 of *LNCS*. Springer.

Li, H.-G., Aghili, S. A., Agrawal, D., and Abbadi, A. E. (2006). FLUX: Fuzzy content and structure matching of XML range queries. In *Proceedings of WWW 2006, May 23-26, 2006, Edinburgh, Scotland*.

Sanz, I., Mesiti, M., Guerrini, G., and Berlanga, R. (2008). Fragment-based approximate retrieval in highly heterogeneous xml collections. *Data Knowl. Eng.*, 64(1):266–293.

Sanz, I., Mesiti, M., Guerrini, G., and Llavori, R. B. (2006). Highly heterogeneous xml collections: How to retrieve precise results?. In (Larsen et al., 2006), pages 232–244.

Schlieder, T. (2002). Schema-driven evaluation of approximate tree-pattern queries. In Springer, editor, *Proceedings of EDBT*.