DISCOVERING VEILED UNSATISFIABLE XPATH QUERIES

Jinghua Groppe and Volker Linnemann
IFIS, University of Lübeck, Lübeck, Germany

Keywords: XML, Query languages, XPath, XPath optimization, XPath satisfiability test.

Abstract: The satisfiability problem of queries is an important determinant in query optimization. The application of a satisfiability test can avoid the submission and the unnecessary evaluation of unsatisfiable queries, and thus save processing time and query costs. If an XPath query does not conform to constraints in a given schema, or constraints from the query itself are inconsistent with each other, the evaluation of the query will return an empty result for any valid XML document, and thus the query is unsatisfiable. Therefore, we propose a schema-based approach to filtering the XPath queries not conforming to the constraints in the schema and the XPath queries with conflicting constraints. We present a complexity analysis of our approach, which proves that our approach is efficient at typical cases. We present an experimental analysis of our prototype, which shows the optimization potential of avoiding the evaluation of unsatisfiable queries.

1 INTRODUCTION

As XML becomes increasingly popular as a language for data storage, automatic exchange and processing, larger and larger as well as more and more data are stored using XML, e.g. the Computer Science Bibliography (University of Trier, 2007) the XML data of which has currently the size 389 Megabytes. Therefore, speeding-up query processing of XML data becomes increasingly important. XPath (W3C, 1999)(W3C, 2003) is a query language for XML data developed by W3C. As well as being a standalone XML query language, XPath is also embedded in other XML languages (e.g. XSLT, XQuery, XLink and XPointer) for specifying node sets in XML documents.

The satisfiability problem of XPath queries is an important issue in XPath evaluation. An XPath query is unsatisfiable if there does not exist any XML document on which the evaluation of the query returns a non-empty result. Therefore, using the satisfiability test can avoid the submission and the unnecessary computation of unsatisfiable XPath queries, and thus saves users' cost and evaluation time. As well as for query optimization, the XPath satisfiablity test is also important in XML access control (Fan et al., 2004), type-checking of transformations (Martens et al., 2004) and XPath-based index update (Hammerschmidt et al. 2005). Therefore, many research efforts focus on the satisfiability test of XPath queries with or without respect to schemas.

In the absence of schemas, the satisfiability test can detect two kinds of errors in an XPath query Q. The first kind of errors is that the structure properties of Q are inconsistent with the XML data model. For example, the XPath query Q1=/following-sibling::a is unsatisfiable, because the root node has no sibling node according to the XML data model. The query Q2=//person/age is tested as a satisfiable XPath query without respect to a schema. However, according to a given schema, e.g. the schema in (Franceschet, 2005), the element person does not have children age. Thus, Q2 is unsatisfiable with respect to the schema. The second kind of errors is that the constraints from Q itself are inconsistent with each other. For example, Q3=a[@v>2][@v<1] is unsatisfiable since @v>2 is contrary to @v<1. Q4=//catgraph/*[parent::*[not(edge)]] is satisfiable because Q4 conforms to the XML data model, and contains no visible conflicting constraints. However, if Q4 is rewritten to /site/catgraph/edge[parent::catgraph[not(edge)] according to a given schema, e.g. one in (Franceschet, 2005), and is further optimized to /site/catgraph[not(edge)]/edge by eliminating reverse axes, then Q4 is unsatisfiable with respect to the schema. (We call Q4 is a query with *hidden* conflicting constraints.) Thus, we can detect more errors in XPath queries if we additionally consider schema information. Therefore, we focus on the satisfiability test of XPath queries in the presence of the schemas

Groppe J. and Linnemann V. (2008).
DISCOVERING VEILED UNSATISFIABLE XPATH QUERIES.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 149-158
DOI: 10.5220/0001684301490158

formulated in the XML Schema language (W3C, 2004a) (W3C, 2004b).

Our schema-based approach first checks whether or not an XPath query Q conforms to the structure, semantics, data type and occurrence constraints given in an XML schema definition S by evaluating Q on S. If Q does not conform to the constraints of S, Q cannot be evaluated completely on S, and thus Q is unsatisfiable. If Q is evaluated completely on S, we rewrite Q to Q' based on the internal data structure generated when evaluating Q on S, which integrates the structure and semantic constraints in S. Q' is equivalent to but contains more information than Q by substituting specific node tests for wildcards, by eliminating redundant parts, by eliminating reverse axes and by substituting non-recursive axes (e.g. *child*) for recursive axes (e.g. *descendant*) whenever possible, and thus can reveal more conflicting constraints. Our approach then checks whether the constraints in Q' are consistent with each other, and filters the queries with conflicting constraints.

Related Work. (Benedikt et al., 2005) theoretically studies the complexity problem of XPath satisfiability in the presence of Document Type Definitions (DTDs), and shows that the complexity of XPath satisfiability depends on the considered subsets of XPath queries and DTDs. We present a practical algorithm for testing the satisfiability of XPath queries. (Hidders, 2003) checks whether the structure properties of XPath queries are consistent with the XML data model. (Lakshmanan et al., 2004) examines the satisfiability test of tree pattern queries with respect to non-recursive schemas. (Kwong and Gertz, 2002) suggests an algorithm for rewriting and the satisfiability test of XPath queries, but allows only non-recursive DTDs and a subset of the XPath axes. We support recursive schemas and all the XPath axes. (Groppe S. et al., 2006) filters unsatisfiable XPath queries by a set of simplification rules, but cannot filter the XPath queries with the hidden conflicting constraints. (Chan et al., 2004) suggests an approach to minimize wildcards in the absence of schemas. We can eliminate wildcards completely in XPath queries. (Olteanu, 2002) eliminates reverse axes in XPath queries according to the axis symmetry of XPath. (Fan et al., 2005) develops an algorithm to rewrite XPath queries to regular XPath queries on recursive DTDs, but only forward axes are considered and the reverse axes and the axes depending on the document order are not allowed.

Our previous contributions (Groppe J. et al., 2006a)(Groppe J. et al., 2006b)(Groppe J. et al., 2006c)(Groppe J. et al., 2007) filter the XPath queries that do not conform to the constraints in an XML Schema definition, but cannot filter the XPath queries with hidden inconsistent constraints. (Groppe J. et al., 2006c) supports a part of the subset of XML Schema supported in this work, and rewrites XPath queries according to schemas in order to refine the given queries rather than to detect the queries with conflicting constraints. In this work, we rewrite the XPath queries that are not detected as unsatisfiable queries to discover possible hidden inconsistent constraints, and apply a set of rules to filter the XPath queries with contradictory constraints.

For XPath, we support all XPath axes, negation operation, and comparison predicates. For XML Schema, we support a significant subset of the XML Schema language, which covers real world schemas and includes e.g. *restriction* and *extension* as well as *all*, *choice* and *sequence* groups. A detailed description on the supported XPath subset and XML Schema subset are given in (Groppe J. et al, 2007). Furthermore, we write DoS for the descendant-or-self axis, AoS for the ancestor-or-self axis, FS for the following-sibling axis and PS for the preceding-sibling axis.

2 XML SCHEMA DATA MODEL

Based on the data model for the XML language in (Wadler, 2000), we develop a data model for XML Schema for identifying the navigation paths of XPath queries on an XML Schema definition. In the model, we write $(x_1,..., x_m)$ for a sequence of entries $x_1,..., x_m$. We use the operator + to concatenate two sequences, e.g. $(x_1) + (y_1, y_2) = (x_1, y_1, y_2)$. Let s be a sequence, we write s[k] for the *k*-th entry of s, and write |s| for the length of s, i.e. the number of entries in s. Furthermore, we also call a node in an XML Schema definition an *XSchema node*.

An XML Schema definition is a set of nodes of type Node. There are four specific Node types in an XML Schema definition, which are associated with *instance element*, *instance attribute*, *instance text* and *instance root* nodes of the XML Schema definition: iElement, iAttribute, iText and iRoot. Accordingly, we define four functions with signature Node→Boolean to test the type of a node: isiElement, isiAttribute , isiText and isiRoot.

Definition 1 (*Instance Nodes*). The *instance nodes* of an XML Schema definition are

- <schema> (which is the *instance root* node)
- <element name=N> (which is an *instance element* node),
- <attribute name=N> (which is an *instance attribute* node),
- attribute node type=T of nodes <element type=T>, which

we denote as @<type=T> (which is an *instance text* node, if T is a built-in simple type),
- <simpleType> (which is an *instance text* node),
- <complexType mixed='true'> (which is an *instance text* node)
- <simpleContent> (which is an *instance text* node) and
- <complexContent mixed='true'> (which is an *instance text* node).

Definition 2 (*Succeeding Node*). A node N2 in an XML Schema definition is a *succeeding node* of a node N1 in the XML Schema definition if
- N2 is a child node of N1, or
- N1=<element type=N> and N2=<simpleType name=N>, or
- N1=<attribute type=N> and N2=<simpleType name=N> , or
- N1=<element type=N> and N2=<complexType name=N>, or
- N1=<element ref=N> and N2=<element name=N>, or
- N1=<attribute ref=N> and N2=<attribute name=N>, or
- N1=<group ref=N> and N2=<group name=N>, or
- N1=<attributeGroup ref=N> and N2=<attributeGroup name=N>, or
- N1=<restriction base=N> and N2=<simpleType name=N>, or
- N1=<extension base=N> and N2=<simpleType name=N>, or
- N1=<extension base=N> and N2=<complexType name=N>.

Figure 1 defines the data model of XML Schema, which consists of a group of functions represented in comprehension notation (Wadler, 2002). The functions child(N) and succeeding(N) relate an XSchema node to a set of XSchema nodes. The functions iChild-helper(N), iChild(N), iAttributeChild(N), iText-helper(N) and iTextChild(N) relate an XSchema node to a set of sequences of XSchema nodes. If $y \in$ iChild(N), then y[1]=N and y[|y|] is an instance child node of N. Other nodes in y are the intermediate nodes visited when searching for y[|y|] of y[1], some of which may be the declaration nodes of model groups, which control the occurrence of y[|y|], and the occurrence order of y[|y|] and its instance sibling nodes in an instance XML document. Taking as example the XML Schema definition city.xsd in Figure 2 in Section 3.1, iChild(D8) = {(D8, D2, D3, D4), (D8, D2, D3, D5)}. iChild-helper(N) returns all the node sequences visited before the instance child nodes and instance attribute nodes of N, e.g. in Figure 2, iChild-helper(D5) = {(D5), (D5, D2), (D5, D2, D3)}. Different from the XML data model, where a node has only one parent node, in XML Schema definitions, a node may have several instance parent nodes. Thus, iPS(x) for finding the instance preceding sibling nodes and iFS(x) for finding the instance following sibling nodes relate a sequence x of nodes to a set of sequences of nodes, where x[1] is the instance parent node of x[|x|]. Let $y \in$ iPS(x), then y[1]=x[1], and y[|y|] is both an instance child node or an instance text node of y[1] and an instance preceding sibling node of x[|x|]. A detailed description on iPS(x), iFS(x) and the data model is given in (Groppe J. et al, 2007).

- child(N) = {N1 | N1 is a child node of N}
- succeeding(N) = {N1 | N1 is a *succeeding* node of N}
- $\text{iChild-helper}(N) = \cup_{i=0}^{\infty} S_i$, where $S_0 = \{(N)\}$,
  $S_i = \{y+(N1) \mid y \in S_{i-1} \wedge N1 \in \text{succeeding}(y[|y|]) \wedge \neg\text{isiElement}(N1) \wedge \neg\text{isiAttribute}(N1)\}$
- $\text{iChild}(N)=\{y+(N1) \mid (y=(N) \wedge \text{isiRoot}(N) \wedge N1 \in \text{child}(N) \wedge \text{isiElement}(N1)) \vee (y \in \text{iChild-helper}(N) \wedge N1 \in \text{succeeding}(y[|y|]) \wedge \text{isiElement}(N1))\}$
- $\text{iAttributeChild}(N) = \{y+(N1) \mid y \in \text{iChild-helper}(N) \wedge N1 \in \text{succeeding}(y[|y|]) \wedge \text{isiAttribute}(N1)\}$
- $\text{iText-helper}(N) = \cup_{i=0}^{\infty} R_i$, where $R_0 = \{(N)\}$,
  $R_i = \{y+(N1) \mid y \in R_{i-1} \wedge N' = y[|y|] \wedge \neg\text{isiText}(N') \wedge \neg\text{isiAttribute}(N') \wedge$ N'≠<complexType> ∧ (N'≠<element type=T> ∨ (N'=<element type=T> ∧ ¬built-in(T))) ∧ N1∈succeeding(N') }
- $\text{iTextChild}(N) = \{y \mid (y \in \text{iText-helper}(N) \wedge \text{isiText}(y[|y|])) \vee (y=z+(N1) \wedge z \in \text{iText-helper}(N) \wedge N' = z[|z|] \wedge \neg\text{isiText}(N') \wedge \text{isiText}(N1) \wedge ((N'=<\text{element type=T}> \wedge N1=\text{attributeNode}(N', \text{type=T})) \vee (N'=<\text{complexType}> \wedge N1 \in \text{succeding}(N')))\}$
- iPS(x) = {y | (y∈iChild(x[1]) ∨ y∈iTextChild(x[1])) ∧ y[|y|]≠@<type=T> ∧ y[y]≠<simpleType> ∧ y[y]≠ <simpleContent> ∧ ((y[|y|]=<complexType mixed='true'> ∨ y[|y|]=<complexContent mixed='true'>) ∨ (x[|x|]=<complexType mixed='true'> ∨ x[|x|]=<complexContent mixed='true'>) ∨ (x=y ∧ ∃i∈{2, 3, ..., |x|}: attribute(x[i], 'maxOccurs')>1) ∨ (∀i∈{1, ..., k}: x[i]=y[i] ∧ x[k+1]≠y[k+1] ∧ k<min(|x|, |y|) ∧ (x[k]=<all> ∨ ∃i∈{2, 3, ..., k}: attribute(x[i], 'maxOccurs')>1 ∨ (y[k+1]<x[k+1] ∧ ∀i∈{2, 3, ..., k}: (x[i]=<sequence maxOccurs=1> ∨ x[i]=<choice maxOccurs=1> ∨ x[i]=<group maxOccurs=1> ∨ (x[i]≠<sequence> ∧ x[i]≠<choice> ∧ x[i]≠<group> ∧ x[i]≠<all>)) ∧ x[k]≠<choice>)))))}
- iFS(x) = {y | (y∈iChild(x[1]) ∨ y∈iTextChild(x[1])) ∧ y[|y|]≠@<type=T> ∧ y[y]≠<simpleType> ∧ y[y]≠ <simpleContent> ∧ ((y[|y|]=<complexType mixed='true'> ∨ y [|y|]=<complexContent mixed='true'>) ∨ (x[|x|]=<complexType mixed='true'> ∨ x[|x|]=<complexContent mixed='true'>) ∨ (x=y ∧ ∃i∈{2, 3, ..., |x|}: attribute(x[i], 'maxOccurs')>1) ∨ (∀i∈{1, ..., k}: x[i]=y[i] ∧ x[k+1]≠y[k+1] ∧ k<min(|x|, |y|) ∧ (x[k]=<all> ∨ ∃i∈{2, 3, ..., k}: attribute(x[i], 'maxOccurs')>1 ∨ (x[k+1]<y[k+1] ∧ ∀i∈{2, 3, ..., k}: (x[i]=<sequence maxOccurs=1> ∨ x[i]=<choice maxOccurs=1> ∨ x[i]=<group maxOccurs=1> ∨ (x[i]≠<sequence> ∧ x[i]≠<choice> ∧ x[i]≠<group> ∧ x[i]≠<all>)) ∧ x[k]≠<choice>)))))}

Figure 1: A data model of XML Schema for evaluating XPath queries on XML Schema definitions.

The auxiliary function attribute(N, 'name') retrieves the value of the attribute 'name' of the node N, which is the name of an element or attribute appearing in an instance document. The function NT: Node×NodeTest→Boolean, which tests an instance XSchema node N against a node test of XPath, is defined as:

- NT(N, label) = (isiElement(N) ∧ attribute(N, 'name')=label) ∨ (isiAttribute(N) ∧ attribute(N, 'name')=label)

• NT(N, *) = isiElement(N) ∨ isiAttribute(N)
• NT(N, text()) = isiText(N) • NT(N, node()) = true

3 EVALUATING XPATH QUERIES

A common XPath evaluator is typically constructed to evaluate XPath queries on XML documents. Our approach evaluates XPath queries on XML Schema definitions rather than on the instance documents of schemas in order to test the satisfiability of XPath queries with respect to schemas. Thus, we name our XPath evaluator *XPath-XSchema* evaluator.

3.1 Schema Paths

Instead of computing the node set of XML documents specified by an XPath query Q, our XPath-XSchema evaluator computes a set of schema paths to the possible resultant nodes, when Q is evaluated by a common XPath evaluator on instance XML documents. If Q cannot be evaluated completely, the schema paths of Q are computed to an empty set of schema paths.

The schema paths are in practice a log of the process of searching for the relevant nodes described by XPath queries from an XML schema definition. In order to better understand the definition of schema paths (see Definition 3), we first outline how the XSchema-XPath evaluator searches for relevant nodes in an XML Schema definition to construct the schema paths. Similar to a common XPath evaluator, our approach starts the search from the root node of the schema. The search continues from an XML Schema node typically to its *succeeding* nodes in the case of a forward axis, or its *preceding* nodes in the case of a reverse axis. The search passes the nodes in the schema, which are not *instance nodes*. The search continues until an *instance* node specified by the current location step is retrieved, and the corresponding node sequence visited is logged in the schema paths.

In the presence of recursive schemas, it may occur that our evaluator revisits a node of the schema without any progress in the processing of the query. We call this a *loop*. For the purpose of detecting a loop, we log the information related with the part of the XPath query, which has been processed. The schema paths for the XPath expressions in a predicate, which are computed in the same way, are attached to the context node of the predicates. We also need a parameter in the schema path to indicate the relation between expressions in a predicate. In an XML Schema definition, an instance node might have several instance parent nodes in that multiple elements might contain some identical sub-elements and each element is declared only once in a schema. Since we cannot retrieve the parent nodes unambiguously from only the XML Schema definition, we need to log the information of the parent nodes in the schema path.

Definition 3 (*Schema Paths*). A schema path the type of which we denote by schema_path is a sequence of pointers to either the schema path records <XP, S, b, z, lp, f>, or the schema path records <o, f>, or schema path records <e> where

- XP is an XPath expression,
- S is a set of sequences of XSchema nodes,
- b is a label and b ∈ {child, parent, FS, PS, self, attribute},
- z is a set of pointers to schema path records,
- lp is a set of schema paths,
- f is a set of sets of schema paths ,
- e is a predicate expression self::node()=C, where C is a literal, i.e. a number or a string, and
- o is a keyword and o ∈ {=, or, and, not}.

Let Q be an XPath query, which is the input of our XPath-XSchema evaluator, and $Q=XP_e/XP_c/XP_r$, where XP_e is the part, which has been evaluated; XP_c is the part, which is being evaluated; XP_r is the part, which has not been evaluated so far by the XPath-XSchema evaluator. In a schema path record, XP is dependent on XP_e. XP is needed for the detection of loop schema paths. S is a set of sequences of XSchema nodes and computed from the XML Schema data model. The last node N_l in each sequence s of S is an instance node, which is visited by the our evaluator when evaluating XP_c, and which is also a context node to compute the following nodes. The first node N_f of s is an instance parent node of N_l, and other nodes in s are ones that are visited when searching for N_l of N_f, some of which may be the nodes of model groups and are useful for consistency checking of occurrence constraints and sequences. b is a label associated with the schema node N_l, indicating an XPath axis, from which the node N_l is generated. b is needed for rewriting. The field z in a schema record R is a set of pointers to the schema path records in which the last schema node of the node sequences is the instance parent node of the last schema node of the node sequences of the record R. Note whenever an instance XSchema node is the first node of a loop, the node has more than one possible instance parent node, and thus there are several sequences of nodes and pointers in a schema path record. lp represents loop schema paths; f represents the schema paths computed from the

predicates that test the last node of S, which is the context node of the predicates. The schema paths can consist of predicate expressions, i.e. {(<self::node()=C>)}. o represents operators like =, or, and and not to indicate the operation on the schema paths of predicates.

Example 1. Our XPath-XSchema evaluator evaluates an XPath query Q = /city//neighbour [name][not(parent::state//city)]/parent::neighbour on the XML Schema definition city.xsd of Figure 2 and computes the schema paths in Figure 3. A detailed, step by step explanation to an example of computing schema paths is given in (Groppe J. et al., 2007).

```
(D1)  <schema>
(D2)   <complexType name='cityT'>
(D3)    <sequence>
(D4)     <element name='name' maxOccurs='1' type='string'/>
(D5)     <element name='neighbour'  maxOccurs='20' type='cityT'/>
(D6)    </sequence>
(D7)   </complexType>
(D8)  <element name='city' maxOccurs='1' type='cityT'/>
(D9) </schema>
```

Figure 2: An XML Schema definition city.xsd.

```
(R1) {(</,  {(/)},  -,  -,  -,  -> ,
(R2)   </city, {(D1, D8)}, child, {R1},  -,  ->,
(R3)    </city/neighbour, {(D8, D2, D3, D5), (D5, D2, D3, D5)}, child, {R2, R3},
(R4)      {(</city/neighbour,  {(D5, D2, D3, D5)}, child, {R3}, -, ->)} ,
(R5)           {{(<-,  {(D8, D2, D3, D5), (D5, D2, D3, D5)}, self, {R2, R3}, -, ->,
(R6)             <name, {(D5, D2, D3, D4)}, child, {R5}, -, ->)},
(R7)          {(<'not', ∅>)}}>
(R8)   <Q, {(D8, D2, D3, D5), (D5, D2, D3, D5)}, parent, {R2, R3}, -, ->) }
```

Figure 3: Schema paths of query Q.

3.2 Computing Schema Paths

We use the semantics technique to describe our XPath-XSchema evaluator, and define the following notations. Let z be a pointer in a schema path and d is a field of a schema path record, we write z.d to refer to the field d of the record to which the pointer z points. Let p be a schema path, then p[k] indicates the *k*-th pointer (or the record to which the *k*-th pointer points) of the schema path p, and |p| be the size of the schema path p, i.e. the number of pointers. Let S be a set of sequences of XSchema nodes, then S(1) indicates an arbitrary sequence of nodes in S. We use the operator / to express the concatenation of two XPath expressions, e.g. XP1/XP2.

The semantics of the XPath-XSchema evaluator is specified by a function L: XPath×schema_path×XPath→Set(schema_path) in Figure 4, which takes two XPath expressions and a schema path as the arguments and yields a set of new schema paths. The first XPath expression is one that is evaluated on a given XML Schema definition in this function, and the second XPath expression is the part XP_e of the given XPath query Q, which has been evaluated so far when the function is called. XP_e is bound to the XP field of a schema path record, and this field is needed for the detection of a loop. The schema path in this function signature is one of the schema paths computed from XP_e. L is defined recursively on the structure of XPath expressions.

For evaluating each location step of an XPath expression, our XPath-XSchema evaluator first computes the axis and the node-test a::n of the location step by iteratively taking the last schema node from a node sequence of the last schema path record from each schema path p in the path set as the context node. For each resultant node r selected by a::n, L first computes a node sequence s based-on the data model of the XML Schema in Figure 1. The function L then constructs a pointer e to a new schema path record, i.e. e→<XP, {s}, b, z, -, -> and extends p to p' by adding the pointer e at the end of the given schema path p, denoted by p'=p+e. If no node is selected by the current location step, the function L computes an empty set of schema paths.

In the case of recursive schemas, a loop is identified whenever the XPath-XSchema evaluator revisits an instance node N of the XML Schema definition without any progress in the processing of the query. In order to avoid an infinite evaluation, we do not continue the evaluation after the node N, once a loop has been detected. We detect loops in the following way: let e=<XP, {s}, b, z, -, -> be a new schema path record generated when computing L(a::n, p, xp). If there exists a record p[k] in p such that S(1)[|S(1)|]=s[|s|] ∧ S=p[k].S ∧ p[k].XP=XP, a loop is detected and the loop path segment is lp = (e, p[k+1], ..., p[|p|]). lp is integrated to the field of the loop schema paths in the schema path record p[k], where the loop occurs. A loop might occur when an XPath query contains the recursive axis descendant, ancestor, preceding or following, which are boiled down to the recursive evaluation of the axis child or parent respectively. For computing L(descendant::n, p, xp), we first compute p_i, where $p_i \in L(child::node(), p_{i-1}, xp) \wedge p_{i-1} \in L(child::node(), p_{i-2}, xp) \wedge \ldots \wedge p_1 = L(child::node(), p, xp)$. If no loop is detected in the path p_i, then let $p_i' = p_i$ and $L^r(self::n, p_i', xp)$ is computed in order to construct a possible new path from p_i. If a loop path segment $(p_i[|p_i|], p_i[k+1], \ldots, p_i[|p_i|-1])$ is detected in the path p_i, then the schema path record $p_i[k]$, from which the loop starts, is modified by integrating the new detected

loop schema path, the new sequence of nodes and the new parent pointer. Note that all the schema paths, which contain the pointer to the schema path record, are also aware of this modification. When a loop is detected, instead of setting $p_i'=p_i$, p_i' is set to empty, i.e. if a loop is detected in p_i, p_i will not contribute to the further computation of schema paths anymore.

The schema paths L(q, fp, -) of a predicate q are added into the field of the predicate schema paths of the record. fp logs the context node of the predicate such that we compute the schema paths of the predicate from fp. When L(q, fp, -) is computed to empty, the main schema paths are computed to an empty set. Checking of data-type and occurrence constraints is presented in (Groppe J. et al., 2007).

- L(e1|e2, p, -) = L(e1, p, -) ∪ L(e2, p, -)
- L(/e, p, -) = L(e, p1, /), where p1=(</, {(/)}, -, -, - >)
- L(e1/e2, p, xp) = {p2 | p2∈L(e2, p1, xp/e1) ∧ p1∈L(e1, p, xp)}
- L(self::n, p, xp) = {p+<xp/self::n, S, self, p[|p|].z, -, -> |
  S=p[|p|].S ∧ NT(S(1)[|S(1)|], n)}
- L(child::n, p, xp) = {p+<xp/n, {s}, child, p[|p|], -, -> |
  NT(s[|s|], n) ∧ S=p[|p|].S ∧ isiElement(S(1)[S(1)]) ∧
  ((s∈iChild(S(1)[|S(1)|]) ∧ n≠text()) ∨
  (s∈iTextChild(S(1)[|S(1)|] ∧ (n=text() ∨ n=node())))}
- L^r(self::n, p, xp) = {p | NT(S(1)[|S(1)|], n) ∧ S=p[|p|].S}
- L(descendant::n, p, xp) = {p' | p'∈$\cup_{i=1}^{\infty}L^r$(self::n, p'_i, xp) ∧ (
($p_i'=p_i$ ∧ p_i∈L(child::node(), p_{i-1}, xp) ∧ ∀k∈{1, …, $|p_i|$-1}: (
p_i[k].XP≠p_i[|p_i|].XP ∨ (S_1(1)[|S_1(1)|]≠S_2(1)[|S_2(1)|] ∧ S_1=p_i[k].S ∧
S_2=p_i[|p_i|].S)) ∧ p_{i-1}∈L(child::node(), p_{i-2}, xp) ∧…∧
p_1∈L(child::node(), p, xp))
∨
($p_i'=\perp$ ∧ (p_i[k]→<p_i[k].XP, p_i[k].S∪ p_i[|p_i|].S, p_i[k].z∪p_i[|p_i|].z,
p_i[k].lp∪{(p_i[|p_i|], p_i[k+1], ..., p_i[|p_i|-1])}, p_i[k].f>) ∧ ∃k∈{1, ..., $|p_i|$-1}: (
p_i[k].XP=p_i[|p_i|]XP ∧ S_1(1)[|S_1(1)|]=S_2(1)[|S_2(1)|] ∧ S_1=p_i[k].S ∧
S_2=p_i[|p_i|].S) ∧ p_i∈L(child::node(), p_{i-1}, xp) ∧
p_{i-1}∈L(child::node(), p_{i-2}, xp) ∧ … ∧ p_1∈L(child::node(), p, xp))))}
- L(parent::n, p, xp) = {p + <xp/parent::n, S, parent, Z1.z, -, -> |
  S=Z1.S ∧ Z1∈p[|p|].z ∧ NT(S(1)[|S(1)|], n) }
- L(ancestor::n, p, xp) = { p' | p'∈$\cup_{i=1}^{\infty}L^r$(self::n, p'_i, xp) ∧ (
($p_i'=p_i$ ∧ p_i∈L(parent::node(), p_{i-1}, xp) ∧ ∀k∈{1, …, $|p_i|$-1}: (
p_i[k].XP≠p_i[|p_i|].XP ∨ (S_1(1)[|S_1(1)|]≠S_2(1)[|S_2(1)|] ∧ S_1=p_i[k].S ∧
S_2=p_i[|p_i|].S)) ∧ p_{i-1}∈L(parent::node(), p_{i-2}, xp) ∧…∧
p_1∈L(parent::node(), p, xp))
∨
($p_i'=\perp$ ∧ (p_i[k]→<p_i[k].XP, p_i[k].S∪ p_i[|p_i|].S, p_i[k].z∪p_i[|p_i|].z,
p_i[k].lp∪{(p_i[|p_i|], p_i[k+1], ..., p_i[|p_i|-1])}, p_i[k].f>) ∧ ∃k∈{1, ..., $|p_i|$-1}: (
p_i[k].XP=p_i[|p_i|]XP∧ S_1(1)[|S_1(1)|]=S_2(1)[|S_2(1)|] ∧ S_1=p_i[k].S ∧
S_2=p_i[|p_i|].S) ∧ p_i∈L(parent::node(), p_{i-1}, xp) ∧
p_{i-1}∈L(parent::node(), p_{i-2}, xp) ∧ … ∧ p_1∈L(parent::node(), p, xp))))}
- L(DoS::n, p, xp) = L(self::n, p, xp) ∪ L(descendant::n, p, xp)
- L(AoS::n, p, xp) = L(self::n, p, xp) ∪ L(ancestor::n, p, xp)
- L(FS::n, p, xp) = {p+<xp/FS::n, {s}, FS, p[|p|].z, -, -> | s∈iFS(s1) ∧
  NT(s[|s|], n) ∧ s1∈p[|p|].S}
- L(following::n, p, xp) = L(AoS::node()/FS::node()/DoS::n, p, xp)
- L(PS::n, p, xp) = {p+<xp/PS::n, {s}, PS, p[|p|].z, -, -> | s∈iPS(s1) ∧
  NT(s[|s|], n) ∧ s1∈p[|p|].S}
- L(preceding::n, p, xp) = L(AoS::node()/PS::node()/DoS::n, p, xp)
- L(attribute::n, p, xp) = {p+<xp/attribute::n, {s}, attribute, p[|p|], -, -> |
  s∈iAttribute(S(1)[|S(1)|]) ∧ NT(s[|s|], n) ∧ S=p[|p|].S}
- L(e[q], p, xp) = {(p'[1], p'[2], …, p'[|p'|-1]) + <p'[|p'|].XP, p'[|p'|].S, p'[|p'|].z, p'[|p'|].lp, p'[|p'|].f∪L(q, fp, -)> | p'∈L(e, p, xp) ∧ L(q, fp, -)≠∅ ∧ fp=(<-, p'[|p'|].S, self, p'[|p'|].z, -, ->)}
- L(e[q_1]…[q_n], p, xp) = {(p'[1], p'[2], …, p'[|p'|-1]) + <p'[|p'|].XP, p'[|p'|].S, p'[|p'|].z, p'[|p'|].lp, p'[|p'|].f∪L(q_1, fp, -)∪…∪L(q_n, fp, -)> | p'=L(e, p, xp) ∧L(q_1, fp, -)≠∅∧…∧L(q_n, fp, -)≠∅∧fp=(<-, p'[|p'|].S, self, p'[|p'|].z, -, ->)}
- L(q_1 and q_2, fp, -) = {(<'and', L(q_1, fp, -)∪L(q_2, fp, -)>) |
  L(q_1, fp, -)≠∅ ∧ L(q_2, fp, -)≠∅}
- L(q_1 or q_2, fp, -) = {(<'or', L(q_1, fp, -)∪L(q_2, fp, -)>) | L(q_1, fp, -)≠∅ ∨ L(q_2, fp, -)≠∅}
- L(q_1 = q_2, fp, -) = {(<'=', L(q_1, fp, -)∪L(q_2, fp, -)>) | L(q_1, fp, -)≠∅ ∧ L(q_2, fp, -)≠∅}
- L(not(q), fp, -) = {(<'not', L(q, fp, -)>)}
- L(q=C, fp, -) = L(q[self::node()=C], fp, -), where q≠self::node()
- L(self::node()=C, fp, -) = {(<self::node()=C>)}

Figure 4: The function L: XPath×schema_path×XPath→Set (schema_path).

3.3 Analyzing Complexity

Different from instance XML documents the topology of which is a tree, an XML Schema definition is a directed graph. In the directed graph leading to the worst-case complexity, each node has directed edges to all nodes. Thus, we assume that in an XML Schema definition S in the worst case, each node in S is an instance node and each node is a succeeding node of all the nodes. In an XPath query Q in the worst case, each location step in Q selects all the instance nodes in S.

Let a be the number of location steps in an XPath query Q. Let N be the number of nodes in an XML Schema definition S. In the worst case, from each schema path p, at most $O(\sum_{k=1}^{N} N!/(N-k)!)$ schema paths are computed with length from |p|+1 to |p|+N, and thus at most $O((\sum_{k=1}^{N}N!/(N-k)!)^a)=O((N!*3)^a)$ schema paths are computed, each of which contains at most O(a∗N) pointers to schema records, for Q. Therefore, the worst case complexity of our approach in terms of run time and space is $O(a*N*(N!*3)^a)$.

The XML Schema definitions of the worst case are rare. A query of the worst case is typically not used. Therefore, it makes sense to investigate the complexity of our approach in typical cases. According to the schema and queries in (Franceschet, 2005), we assume that the typical cases are characterized as follows: each node in an XML Schema definition S has only a small number of succeeding nodes compared with the number N of nodes in S; for each location step in the XPath query Q, the number of nodes visited is on the average less

than a constant C, and thus less than C schema paths are computed for each location step. Therefore, the complexity of runtime and space of our approach is O(a∗N∗C) for the typical cases.

4 REWRITING XPATH QUERIES

If an XPath query Q is computed to a non-empty set of schema paths by our evaluator on an XML Schema definition, the XPath query is only *maybe* satisfiable, since the satisfiability test in the supported subset of XPath is undecidable (Benedikt, et al., 2005) and our evaluator does not check whether or not two or more location steps in Q contradict each other. In this section, we present filtering the queries with conflicting constraints by rewriting the queries to the empty expression ⊥ based on their schema paths.

4.1 Mapping Schema Paths to (Regular) XPath Queries

The function M(L) in Figure 5 maps a set of schema paths $L=\{p_1, ..., p_m\}$ to an XPath query Q'. The function M(p) maps a schema path $p=(r_1, ..., r_n)$ to a sub-expression e of the query Q'. The function M(r) maps a schema path record r to a pattern of the sub-expression e. The patterns are concatenated in order with '/' to form the sub-expression $e=M(p)=M(r_1)+'/'+...+'/'+M(r_n)$, where we use '+' to denote concatenation of strings. Disjunctions of the sub-expressions form the mapped query $Q'=M(L)=M(p_1)+' | '+...+' | '+M(p_n)$. In order to compute a pattern from a schema path record <XP, S, b, z, lp, f>, <o, f> or <e>, we need the following functions: location(S, b) computes the axis and the node-test of a pattern; loops(lp) computes the union of loop patterns. Let us assume that B is a pattern, then we define B* as a loop pattern, in which the Kleene star denotes an arbitrary repetition of the pattern B. As an example, if B=a, then B*=(⊥ | a | a/a | a/a/a |...).

Let L be a set of schema paths, p be a schema path and r be a schema path record, such that $L=\{p_1,..., p_m\}$ and $p=(r_1,..., r_n)$, where $p \in L$. The semantics of the mapping function M, which maps a set of schema paths to a (regular) XPath expression, is defined in Fig 6. Note that in the mapping functions of Fig. 6, the two fields XP and z in a schema path record r are left out since they do not contribute to the computation of the mapping.

If we use the function M^r(<S, b, lp, ->), we get a regular XPath expression with loop patterns using the Kleene star ∗, which is not a standard XPath operator; if we use the function M(<S, b, lp, ->), we get a standard XPath expression without loop patterns.

- $M(L) = M(p_1)+' | '+ ...+' | '+M(p_m)$
- $M(p) = M(r_1)+M(r_2)+'/'+...+'/'+M(r_n)$, if $N='/' \wedge N=S(1)[|S(1)|] \wedge S= r_1.S$
- $M(p) = M(r_1)+'/'+...+'/'+M(r_n)$, if $N\neq'/' \wedge N=S(1)[|S(1)|] \wedge S= r_1.S$
- M(<S, b, -, ->) = location(S, b)
- $M(<S, b, -, \{L_1, ..., L_n\}>) = M(<S, b, -, ->)+'['+M(L_1)+']'+...+'['+M(L_n)+']'$
- M (<S, b, lp, ->) = 'descendant::'+attribute(N, 'name'), where b='child' ∧ N=S(1)[|S(1)|]
- M (<S, b, lp, ->) = 'ancestor::'+attribute(N, 'name'), where b='parent' ∧ N=S(1)[|S(1)|]
- M^r(<S, b, lp, ->) =loops(lp)+location(S,b)
- $M(<S, b, lp, \{L_1,..., L_n\}>) = M(<S, b, lp, ->)+'['+M(L_1)+']'+...+'['+M(L_n)+']'$
- $M^r(<S, b, lp, \{L_1,..., L_n\}>) = M^r(<S, b, lp, ->)+'['+M(L_1)+']'+...+'['+M(L_n)+']'$
- M(<'not', {L}>) = 'not'+'('+ M(L)+')'
- $M(<'or', \{L_1, L_2\}>) = M(L_1)+' or '+M(L_2)$
- $M(<'=', \{L_1, L_2\}>) = M(L_1)+' = '+M(L_2)$
- M(<self::node()=C>) = 'self::node()=C'
- location (S, -) = '/', where S(1)[|S(1)|]= '/'
- location(S, b) = b+'::'+attribute(N, 'name'), where (isiElement(N) ∨ isiAttribute(N)) ∧ N=S(1)[|S(1)|]
- location(S, b) = b+'::text()', where isiText(N) ∧ N=S(1)[|S(1)|]
- location(S, b) = b+'::node()' where N= '/' ∧ N=S(1)[|S(1)|]
- $loops(lp) = loops(\{p_1,..., p_k\})='((('+M(p_1)+'/)*'+' | '...' | ' + '('+M(p_k)+'/)*)'$

Figure 5: Functions mapping schema paths to a (regular) XPath expression.

Proposition 1. Let L be a set of schema paths, Q^r be the regular XPath expression mapped from L, and Q be the standard XPath expression mapped from L. The evaluation of Q returns the same node set as Q^r for any valid XML document (Groppe J. et al., 2006c).

4.2 Optimizing Mapped XPath Queries

The mapped XPath query can be optimized by eliminating redundant parts, reverse axes and recursive axes. For this optimization, we develop a set of rewriting rules. Different from the rewriting rules in (Olteanu et al., 2002), which eliminates reverse axes based on the symmetry of the XPath axes, we eliminate reverse axes mainly based on the symmetry of the schema paths. The reverse axes, which are remaining after eliminating redundant parts, can be eliminated using the ruleset in (Olteanu et al., 2002).

Let a be an axis, n be a nodetest, e be a pattern and q be a qualifier. The rewriting rules, which eliminate reverse axes and redundant parts in the XPath expression mapped from a set of schema paths, are defined as follows.

- e/attribute::n1/parent::n2[q] ≡ e[q][attribute::n1]
- e/child::n1/parent::n2[q] ≡ e[q][child::n1]

- e1/child::n1/e2/parent::n3[q] ≡ e1[q][child::n1/e2], where e2 contains only the axes FS and PS
- e/attribute::n1[parent::n2[q]] ≡ e1[q]/attribute::n2
- e1/child::n1[parent::n2[q]] ≡ e1[q]/child::n1
- e1/child::n1/e2[parent::n3[q]] ≡ e1[q]/child::n1/e2, where e2 contains only the axes FS and PS
- e1[attribute::n1/parent::n2[q]] ≡ e1[q][attribute::n1]
- e1[child::n1/parent::n2[q]] ≡ e1[q][child::n1]
- e1[child::n1/e2/parent::n3[q]] ≡ e1[q][child::n1/e2], where e2 contains only the axes FS and PS
- e/self::n[q] ≡ e[q]
- e[q][q] ≡ e[q]
- e[q]/q ≡ e/q
- e[true()] ≡ e
- [not(false())] ≡ [true()]
- [q or true()] ≡ [true()]
- [q or false()] ≡ [q]
- [q and true()] ≡ [q]
- e*/parent::n ⊆ ancestor::n
- e*/child::n ⊆ descendant::n

Note that in the rules, e*/child::n is the pattern mapped by M^r[<S, b, lp, ->] and descendant::n is the pattern mapped by M[<S, b, lp, ->], when b='child'. As shown in Proposition 1, although descendant::n retrieves a superset of the node set retrieved by e*/child::n, the entire XPath query returns the same node set for all valid XML documents when using either descendant::n or e*/child::n.

4.3 Filtering XPath Queries with Conflicting Constraints

We apply the rules in Figure 6 to the queries rewritten from schema paths to filter the queries, which contain conflicting constraints. Although the rule set can be directly applied to given queries, application of the rules to the rewritten queries, which exclude redundant parts, wildcards, reverse axes and recursive axes, can filter more unsatisfiable queries.

Let e (e1, e2… respectively) be an XPath expression. If a sub-expression of an XPath query is reduced to the empty expression ⊥, the XPath query is reduced to ⊥.

- ⊥ | ⊥= ⊥
- e/⊥ = ⊥
- ⊥/e = ⊥
- e[⊥] = ⊥
- ⊥[e] = ⊥
- ⊥ and e = ⊥
- ⊥ or ⊥=⊥
- e1[not(e_2)]/e_2=⊥
- e1[not(e2)][e2]=⊥
- e1[not(e2)][e2/e3]=⊥
- e[@t=c1][@t=c2]=⊥ if c1≠c2
- e[@t<c1][@t=c2]=⊥ if c1≤c2
- e[@t<c1][@t>c2]=⊥ if c1≤c2
- e[@t<c1][@t≥c2]=⊥ if c1≤c2
- e[@t≤c1][@t=c2]=⊥ if c1<c2
- e[@t≤c1][@t>c2]=⊥ if c1≤c2
- e[@t≤c1][@t≥c2]=⊥ if c1<c2

Figure 6: Rules for filtering queries with conflicting constraints.

5 PERFORMANCE ANALYSIS

We have implemented a prototype of our approach in order to verify the correctness of our approach and to demonstrate the optimization potential by avoiding the evaluation of unsatisfiable XPath queries. (Groppe J. et al., 2007) presents a comprehensive performance analysis on detecting the unsatisfiable XPath queries that do not conform to the constraints in an XML Schema definition, i.e. the schema paths of the queries are computed to the empty set, and experimental results show that our approach can achieve a speedup up to several orders of magnitudes over common XPath evaluators when detecting unsatisfiable XPath queries. Therefore, this performance analysis focuses on the unsatisfiable XPath queries, which conform to the constraints imposed by a schema, but contain hidden conflicting constraints. Our approach first computes the schema paths of the queries by evaluating the queries on an XML Schema definition, then rewrites these queries based on the schema paths in order to make hidden conflicting constraints visible, and finally applies the rules to the rewritten queries to filter the queries with conflicting constraints. We study the detection of the unsatisfiable XPath queries by our approach and the evaluation of these unsatisfiable queries by common XPath evaluators.

The test system for all experiments is an Intel Core 2 CPU T5600 processor, where we disable one CPU, 1.83 Gigahertz with 2 Gigabytes RAM, Windows XP as operating system and Java VM version 1.6.0. We use the XQuery evaluators Saxon version 8.0 (//saxon.sourceforge.net) and Qizx version 0.4pl (//www.xfra.net/quizxopen) to evaluate the XPath queries on XML data. We use the XPathMark benchmark (Franceschet, 2005) as the source of our experimental data, and generate data from 0.116 Megabytes to 11.597 Megabytes by using the data generator of (Franceschet, 2005). An XML Schema definition benchmark.xsd (Groppe J. et al., 2007) is manually adapted according to the DTD benchmark.dtd (Franceschet, 2005) and the instance documents in order to integrate as many constructs of the XML Schema as possible and to specify more specific data types for values of elements and attributes, which are all declared as #PCDATA in benchmark.dtd.

The queries Q1-Q15 in Table 1 conform to the semantics, structure, data-type and occurrence constraints given in benchmark.xsd, but contain hidden conflicting constraints. Thus, the schema paths of these queries are computed to a non-empty set. Queries Q1'-Q15' in Table 1 are the rewriting of queries Q1-Q15 based on their schema paths. The

rewritten queries disclose the hidden conflicting constraints. Furthermore, the queries Q1-Q15 are also designed to contain as many constructs of the XPath language as possible in order to test how the different constructs of the XPath language influence the processing performance. We present the average results of ten executions of these queries.

Table 1: Queries Q1-Q15 and rewritten queries Q1'-Q15'.

Original and rewritten Queries	
Q1	/site/catgraph[not(edge)]/∗
Q1'	/site/catgraph[not(edge)]/edge
Q2	/site/catgraph[not(edge)]/self::node()/∗
Q2'	/site/catgraph[not(edge)]/edge
Q3	/site/regions/europe[(@area or ∗/name) and not(item)]
Q3'	/site/regions/europe[item/name][not(item)]
Q4	/site/regions/europe/∗[parent::∗[not(item)]]
Q4'	/site/regions/europe[not(item)]/item
Q5	//europe/∗[parent::∗[not(item)]]
Q5'	/site/regions/europe[not(item)]/item
Q6	/site/closed_auctions/closed_auction/buyer [@∗][not(@person)]
Q6'	/site/closed_auctions/closed_auction/buyer [@person][not(@person)]
Q7	/site/closed_auctions/closed_auction/buyer[@∗]/ self::∗[not(@person)]
Q7'	/site/closed_auctions/closed_auction/buyer [@person][not(@person)]
Q8	//buyer[@∗][not(@person)]
Q8'	/site/closed_auctions/closed_auction/buyer [@person][not(@person)]
Q9	/site/people/person/profile[@∗>50][@income<10]
Q9'	/site/people/person/profile[@income>50][@income<10]
Q10	/site/people/person/profile[@∗>50]/interest /parent::∗[@income<10]
Q10'	/site/people/person/profile[@income>50][@income<10] [interest]
Q11	/site/people/person/profile[@∗>50][@∗<99][@income<10]
Q11'	/site/people/person/profile[@income>50][@income<99] [@income<10]
Q12	/site/people/person/profile[@∗>50][@∗<99][@∗>30] [@income<10]
Q12'	/site/people/person/profile[@income>50][@income<99] [@income>30][@income<10]
Q13	/site/people/person/profile[@∗>50][@∗<99][@∗>30] [@∗>40][@income<10]
Q13'	/site/people/person/profile[@income>50][@income<99] [@income>30][@income>40][@income<10]
Q14	//profile[@∗>50][@income<10]
Q14'	/site/people/person/profile[@income>50][@income<10]
Q15	//profile[@∗>50][@∗<99][@∗>30][@income<10]
Q15'	/site/people/person/profile[@income>50][@income<99] [@income>30][@income<10]

Figure 7 presents the time of filtering the unsatisfiable queries Q1-Q15 by our approach, consisting of three times: the time of computing schema paths, i.e. evaluating Q1-Q15 on benchmark.xsd; the time of rewriting Q1-Q15 based on the schema paths, i.e. mapping schema paths to an XPath query Q and optimizing Q by the rules in Section 4.2; the time of filtering XPath queries with conflicting constraints by the rules in Section 4.3. The overhead of filtering unsatisfiable queries is mainly evaluating XPath queries on the schema. Among 15 queries, Q5, Q8, Q14 and Q15 are queries with recursive axes, which we call recursive queries; others do not contain recursive axes, which we call non-recursive queries. Non-recursive queries can be evaluated very fast and are on the average 7.2 faster than the recursive queries. The overhead of rewriting and rule application is very low. The time of rewriting and rule application is 32.6% of the time of computing schema paths for the non-recursive queries, the time ratio is 2.6% for the recursive queries, and the time ratio is 11% for all the queries.

Figure 8 and Figure 9 present the speedup achieved by our approach over Saxon and Qizx when the evaluation of Q1-Q15 returns an empty result. The results show that our approach can detect unsatisfiable queries efficiently. At a data size of 6 Megabytes, our approach is 199 times (and 39.6 times) faster on the average when evaluating the non-recursive queries, and 35.6 times (and 10 times) faster on the average when evaluating the recursive queries, than Saxon (and Qizx). At a data size of 12 Megabytes, our approach is 392 times (and 80 times) faster on the average when evaluating the non-recursive queries, and 69.5 times (and 20 times) faster on the average when evaluating the recursive queries, than Saxon (and Qizx).

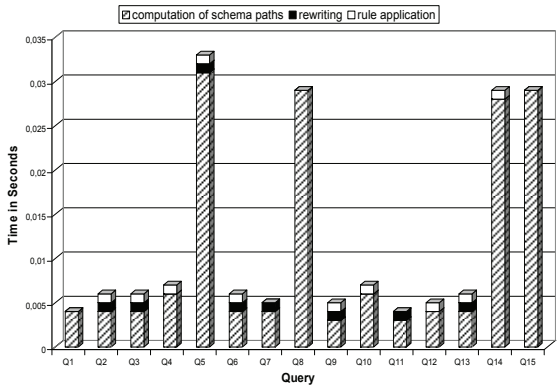

Figure 7: Filtering queries Q1-Q15 by our approach.

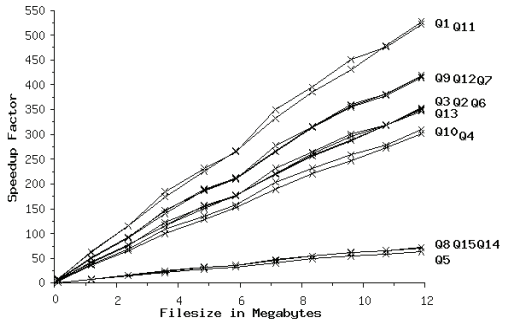

Figure 8: Speedup by our approach over Saxon when evaluating Q1-Q15.

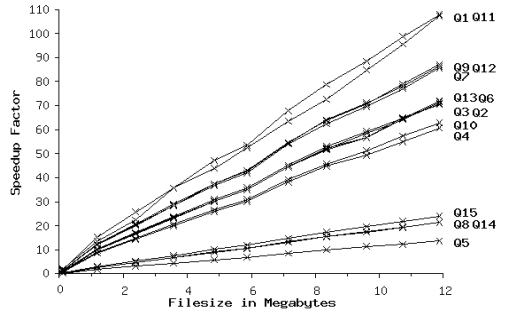

Figure 9: Speedup by our approach over Qizx when evaluating Q1-Q15.

6 CONCLUSIONS

We have proposed a data model for the XML Schema language, which identifies the navigation paths of XPath queries on XML Schema definitions. Based on the data model, we have developed an XPath-XSchema evaluator, which evaluates XPath queries on an XML Schema definition in order to filter the queries not conforming to the constraints imposed by the schema and in order to rewrite queries. When an XPath query does not conform to the constraints in the schema, our evaluator computes an empty set of schema paths, i.e. the XPath query is unsatisfiable. If a non-empty set of schema paths is computed for an XPath query, we rewrite the query from its schema paths, and apply the rules of conflicting constraints to the rewritten queries to further filter the queries with conflicting constraints.

The experimental results of our prototype show that the application of our approach can significantly optimize the evaluation of XPath queries by filtering unsatisfiable XPath queries. A speed-up factor up to several orders of magnitudes is possible.

REFERENCES

Benedikt, M., Fan, W., Geerts, F., 2005. XPath satisfiability in the presence of DTDs. In PODS'05.

Chan, C.Y., Fan, W., Zeng, Y, 2004. Taming XPath queries by minimizing wildcard steps. In VLDB'04.

Fan, W., Chan, C., Garofalakis, M., 2004. Secure XML querying with security views. In SIGMOD'04.

Fan, W., Yu, J.X., Lu, H., Lu, J., Zeng, Y., 2005. Query translation from XPath to SQL in the presence of recursive DTDs. In VLDB'05.

Franceschet, M., 2005. XPathMark – An XPath benchmark for XMark. Research report PP-2005-04, University of Amsterdam.

Groppe, S., Böttcher, S., Groppe, J, 2006. XPath query simplification with regard to the elimination of intersect and except operators. In XSDM'06 in association with ICDE'06.

Groppe, J., Groppe, S., 2006a. Filtering unsatisfiabile XPath queries. In ICEIS'06.

Groppe, J., Groppe, S., 2006b. A prototype of a schema-based XPath satisfiability tester. In DEXA'06.

Groppe, J, Groppe, S., 2006c. Satisfiability-test, rewriting and refinement of users' XPath queries according to XML Schema definitions. In ADBIS'06.

Groppe, J., Groppe, S., 2007. Filtering unsatisfiable XPath queries. Data Knowl. Eng. 64(1):134 – 169.

Hammerschmidt, B.C., Kempa, M., Linnermann, V., 2005. The index update problem for XML data in XDBMS. In ICEIS'05.

Hidders, J., 2003. Satisfiability of XPath expressions. In DBPL'03.

Kwong, A., Gertz, M. 2002. Schema-based optimization of XPath expressions. Techn. Report, University of California.

Lakshmanan, L., Ramesh, G., Wang, H., Zhao, Z., 2004. On testing satisfiability of tree pattern queries. In VLDB'04.

Martens, W., Neven, F., 2004. Fronties of tractability for typechecking simple XML transformations. In VLDB'04.

Olteanu, D., Meuss, H., Furche, T., Bry, F., 2002. XPath: looking forward. XML-Based Data Management (XMLDM), EDBT Workshops.

University of Trier, 2007. Computer Science Bibliographie. dblp.uni-trier.de/, 17th July 2007.

Wadler, P., 2002. Two semantics for XPath. Tech. Report.

W3C, 2004a. XML Schema part 1: Structures second edition. W3C Recommendation. www.w3.org/TR/xmlschema-1.

W3C, 2004b. XML Schema part 2: Datatypes second edition", W3C Recommendation. www.w3.org/TR/xmlschema-2.

W3C, 1999. XPath version 1.0. W3C Recommendation. www.w3.org/TR/xpath/.

W3C, 2003. XPath Version 2.0. W3C Working Draft. www.w3.org/TR/xpath20/.