Improving the Efficiency of XPath Execution on Relational Systems

Haris Georgiadis and Vasilis Vassalos

Department of Informatics
Athens University of Economics and Business
Athens, Greece
{harisgeo,vassalos}@aueb.gr

Abstract: This work describes a method for processing XPath on a relational back-end that significantly limits the number of SQL joins required, takes advantage of the strengths of modern SQL query processors, exploits XML schema information and has low implementation complexity. The method is based on the splitting of XPath expressions into Primary Path Fragments (PPFs) and their subsequent combination using an efficient structural join method, and is applicable to all XPath axes. A detailed description of the method is followed by an experimental study that shows our technique yields significant efficiency improvements over other XPath processing techniques and systems.

1 Introduction

In the past few years the adoption of XML for a variety of roles in e-business applications has increased significantly and continues to increase. XML is increasingly used as a data exchange/messaging format between applications or Web services [22], as a data model for middleware-based data integration [23] and as a data model for storing and querying application data [24]. Given the growing importance and presence of XML data, the need to query and maintain them arises in most of the above cases. At the same time, business applications as always rely heavily on agreed-upon schemas and descriptions for data modeling (in the case of XML, XML Schema [25] or DTD). XML storage and query systems fall into three main categories:

- Native XML systems [29,30] that use storage models indexing and querying mechanisms specially designed for XML data. Storage models are based on path sequences, flat files [31], tree-based node clustering [Natix] or other techniques.
- XML-shredding systems [2,3,4,5,18] that decompose XML documents into relations, store them in RDBMSs and process them using RDBMS machinery.
- Hybrid approaches that store XML as CLOBs/BLOBs into relational tables, either exclusively or in combination with shredding [26].

XML shredding techniques can be schema-oblivious, where the relations into which XML is translated are fixed irrespective of the XML document structure, as in the Edge mapping [1]. There, all element nodes are stored as tuples in a single central relation. Alternatively, shredding can be schema-aware [4,5], where the relational schema constructed is adapted to the XML schema information available.

Because of the wide availability, robustness and manageability of RDBMSs, the shredding and hybrid solutions have received a lot of attention. Several techniques and systems have been proposed [2,3,11,12] for SQL-based XML processing, supporting large subsets of well-known XML query languages, namely XQuery[7] or XPath[6]. These systems and techniques translate expressions of these languages into SQL equivalents and execute them on relational back ends. In earlier attempts, e.g., [4], the SQL translations had a large number of foreign-key joins, usually proportional to the number of steps in paths. This technique was unable or inefficient to handle a series of XPath features, such as the descendant '//' and several other axes, recursion and wildcards ('*'). Several techniques have been proposed to tackle the above problems. For schema-oblivious mappings, efficient methods such as region encoding [2] and dewey encoding [9] have been proposed to encode both structural relationships among elements and ordering information, and to transform structural relationships such as "descendant" into range comparisons. These techniques, by themselves, do not accelerate simple path traversals: again the number of joins is proportional to the number of steps. Regarding schema-aware mapping, for example, schema information can has been used to eliminate redundant joins [11], whereas recursive queries can be handled using the recursion capabilities of SQL99 [12].

This work describes a novel XPath processing approach that yields significant performance benefits while being quite easy to implement and combine with existing techniques. A key novel concept of our approach is the Primitive Path Fragment (PPF), which is a syntactic unit of an XPath expression. PPFs can be efficiently evaluated in a holistic fashion using a root-to-node path index and regular expression matching, to eliminate the need for structural joins. We describe Primitive Path Fragments and their processing in Section 4.3. The second important part of our approach is a method based on the properties of Dewey encoding [9] for efficiently performing the necessary structural joins between PPFs. Our implementation of Dewey encoding, its properties and its use for joining PPFs are described in Section 4.2. The complete XPath to SQL translation algorithm is presented in Section 4.3.

PPF-based XPath processing can be applied both to schema-oblivious and schema aware XML shredding. In schema-aware shredding, data are apportioned into several relations. The existence of schema information and its utilization in the translation, allows for optimizations, such as avoiding redundant root-to-node path filtering, as discussed in Section 4.5. The experimental evaluation in Section 5.1 confirms the benefits of applying the PPF-based processing in conjunction with schema-aware XML shredding. Hence this work focuses on such a translation scheme, describing it briefly in Section 3. Our implementation of PPF-based processing is built on top of an Oracle 10g-based XML management system using schema-aware XML shredding. In our experimental study, which is in Section 5, we are comparing our technique to the latest version of MonetDB/XQuery, our implementation of XPath accelerator on top of Oracle 10g, and the built-in XPath processor of a major commercial RDBMS, on a large number of representative XPath queries on different data sets. We discuss related work in Section 6 and present our conclusions in Section 7.

In summary, we show how PPF-based XPath processing can handle efficiently a large subset of XPath 2.0 that includes all XPath axes, path union, nested expressions, and logical, arithmetic and position predicates. PPF-based XPath processing offers a comprehensive solution to the problem of XPath processing that exploits the

strengths of relational query processing and optimization with minimal tuning and gives significant performance gains over existing techniques with much less implementation complexity.

2 Background

2.1 XML data model, XML Schema and XPath

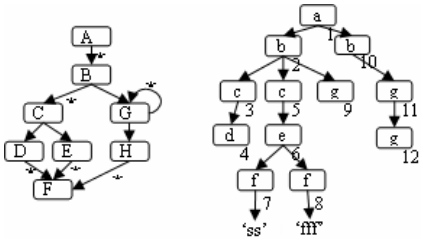
An XML document can be represented as a rooted, ordered, labeled tree, where each node corresponds to an element or a text value. The edges represent (direct) element-subelement or element-value relationships. Tags, IDs, IDREFs and other attributes are modeled by node labels consisting of a set of attribute-value pairs. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a preorder traversal of the tree nodes. Figure 1(b) shows the tree representation of an XML document, where the numbers outside the nodes represent node identifiers.

The structure of an XML document can be described by an XML Schema. An XML Schema can be represented as a directed graph [12], where vertices correspond to element definitions and edges represent nesting relationships. A simple XML Schema graph is illustrated in Figure 1(a). An element node in a document described by an XML Schema instantiates a particular type defined in the schema.

XPath [6] is a language for locating XML nodes. The main construct of XPath is the path expression which consists of a sequence of steps, separated with the '/' character, to address nodes within the XML representation of an XML document. Each step has three parts: an *axis*, such as *child*, *parent* and *descendant*, which defines the structural relationship of nodes to be selected with respect to those selected by the preceding step, a *node test* which defines the name or the kind of nodes to be selected and, optionally, one or more *predicates* which set further restrictions to the nodes to be selected. Wildcards ('*') can be used as node tests that select nodes regardless of their name. For example, the XPath expression '/A/*[C//F=2]' returns elements that are children of element 'A', and have at least a child element 'C', which has at least one descendant element named 'F' with text value equal to 2.

2.2 XML element position representation

A key issue for efficient XML processing is an appropriate representation of the positions of XML elements. In order to preserve the document order of elements and also to test more directly the structural relationship among nodes, we use dewey en-coding [9]. Dewey encoding assigns to each node a vector that represents the path from the document's root to the node. Each component of the path represents the local order of an ancestor node. The dewey encoding for each element of Figure 1(a) is shown in Figure 1(c). Dewey encoding, like other positional encodings such as region encoding [9] and ORDPATH [19], allows the transformation of structural relationships, such as *descendant* or *sibling*, into number or string comparisons of the encodings. For example, a tree node n encoded as $n_1.n_2....n_k$ is a descendant of tree node m encoded as $m_1.m_2...m_f$ *iff* $k>f$ and $n1.\ n_1.n_2....n_f = m_1.m_2...m_f$.



id	par	dewey position	XML schema Type
1		1	A
2	1	1.1	B
3	2	1.1.1	C
4	3	1.1.1.1	D
5	2	1.1.2	C
6	5	1.1.2.1	E
7	6	1.1.2.1.1	F
8	6	1.1.2.1.2	F
9	2	1.1.3	G
10	1	1.2	B
11	10	1.2.1	G
12	11	1.2.1.1	G

(a) Graph representation of an XML Schema

(b) Sample XML document conforming to Schema

(c) XML element descriptors

Fig. 1.

3 XML Schema-To-Relational Mapping

In order to represent XML elements in relational structures, we have defined 4 descriptors that characterize all element nodes, as shown in Figure 1(c). Apart from node id and Dewey position, mentioned earlier, we also keep a node's parent id. Moreover, we associate with each element a root-to-node path id. Path ids and their use as an index are described in the next Section. We describe below how these descriptors as well as text and attribute values are stored in relational structures.

Even though PPF-based processing can be used effectively with schema-oblivious XML shredding, as we will see in Section 5, it yields greater benefits used on top of a schema-aware XML to relational translation, and this is what we focus on.

Our system takes as input the XML Schema's graph representation and creates the respective relational structures according to a fixed set of mapping rules, where:

- each complex type is mapped into a separate relation,
- each element definition is also mapped into a separate relation, unless it is of a globally defined, already mapped complex type,
- text and attribute nodes of an element are mapped into columns of the appropriate type in the element's corresponding relation

Each relation has a primary key 'id' column that stores the element id, and one or more foreign key columns referring to all possible parent relations, for storing element nesting relationships. Note that, in case of recursive schemata where a complex type contains elements of the same complex type, the corresponding relation maintains a foreign key relationship to itself. Relations that correspond to document elements have an additional column, named 'doc_id', to distinguish documents from one another. Finally, Dewey position is stored in the 'dewey_pos' column as a binary string. The specific encoding and its properties are discussed in Section 4.2.

Note that our mapping scheme does not use inlining [5], namely, the mapping of certain element definitions into columns instead of separate relations. This technique

is mainly used to reduce the total number of relations and, subsequently, the number of structural joins in the SQL translations. As we discuss in section 4, PPF-based processing eliminates many of these joins in a different effective fashion.

3.1 Root-to-node path index and other relational indices

We store for each element node its root-to-node path and use it as an index. Since, for a typical set of XML documents conforming to an XML schema, the total number of distinct paths is expected to be much smaller than the total number of nodes, all paths are stored in a separate relation, named 'Paths'. All mapping relations maintain a foreign key reference to this relation, in a column named 'path_id'. The 'Paths' relation is filled gradually during insertions: when an element is to be inserted, its path will be inserted in the 'Paths' relation, as long as it hasn't been already inserted during a previous element insertion. As we discuss further in Section 4.1, *Primitive Path Fragments* of an XPath query can be handled by applying simple regular expression filtering over root-to-node paths, which significantly reduces the number of structural joins in the final SQL statement.

For each relation, the following relational indexes are also created and maintained: an index for the 'id' column, one index for each parent foreign-key column and one concatenated (composite) index on columns dewey_pos and path_id. In our current implementation, all indices are created as standard B-trees and hence the overall cost.

4 XPath-To-SQL Translation

This section describes PPF-based XPath processing. A key novel concept of our approach is the *Primitive Path Fragment* (PPF), which is a syntactic unit of an XPath expression. PPFs can be efficiently evaluated in a holistic fashion using a root-to-node *path index* and *regular expression matching,* to eliminate the need for structural joins. In particular, a PPF can be handled by a natural join of a single relation with the 'Paths' relation, followed by an appropriate restriction in the 'where' clause of the SQL statement. This restriction filters the root-to-node paths against a regular expression derived from the step sequence of the fragment. We describe Primitive Path Fragments and their processing in the next section.

The hierarchical relationship of each consecutive pair of such fragments is handled by theta-joining the two relations with appropriate lexicographic comparison between their dewey_pos columns. We describe the method for combining PPFs in Section 4.2. The complete XPath to SQL translation algorithm is presented in Section 4.3, while additional optimizations are described in Sections 4.4 and 4.5.

4.1 Identifying and Processing Primitive Path Fragments

Let's suppose we want to translate the XPath expression '/A/B/C/*/F' into an equivalent SQL statement for documents conforming to the XML schema shown in Figure 1(a). Taking into account the graph representation of the schema and its relational

mapping, it is easy to conclude that F is the only relation that could potentially store the 'F' elements defined by the given XPath expression. Each tuple in 'F' is assigned a path-id number referring to a certain root-to-node path in the 'Paths' relation. Therefore, the tuples corresponding to the required 'F' elements can be retrieved with a single SQL select statement that joins relations 'F' and 'Paths' and adds a restriction to the 'path' column of the 'Paths' relation so as to match the path '/A/B/C/*/F'. SQL's *LIKE* operator, in combination with string manipulation functions, could handle such simple pattern matching. To deal with more complex patterns, as are many XPath expressions, we translate the path into a simple regular expression and then use a regular expression filtering function, within the SQL statement, to perform the matching. Several commercial RDBMSs (e.g., mySQL, Oracle 10g) have incorporated such functions into their function library. We use the *REGEXP_LIKE* function of Oracle 10g which follows the exact Extended Regular Expression (ERE) syntax and semantics defined in the POSIX [17].

Path id filtering helps us handle certain sequences of steps in an XPath expression. The steps of such a sequence must

- all have only forward axes or only backward axes and,
- they must not have predicates, except for the last step.

We call such paths *Forward Simple Paths* and *Backward Simple Paths* respectively. Table 1 illustrates several forward and backward simple paths and their corresponding regular expression equivalents.

Table 1. Examples of mapping forward or backward paths into regular expressions

Forward or Backward Path	Regular expression
//B/C	'^.*/B/C$'
/A/B//F	'^A/B/(.+/)?F$'
//C/*/F	'^.*/C/[^/]+/F$'
/parent::F/ancestror::B/parent::A	'^.*/A/B/(.+/)?F$'

More specifically, we divide the main path of an XPath expression which we call '*Backbone Path*', as well as the paths included in predicates, into fragments, named '*Primitive Path Fragments*' (PPFs).

Definition: We call '*Primitive Path Fragment*', a sequence of one or more consecutive steps of an XPath expression for which one of the following is true:

a) It is a *forward simple path*

b) It is a *backward simple path*

c) It is a single step whose axis is one of the following*: following, following-sibling, preceding or preceding-sibling*

Recall that a forward or backward simple path can have predicate(s) only in the last step, thus a predicate in an intermediate step of a forward or backward path always separates the path into two PPFs. Our PPF-processing system parses the XPath expression and creates a corresponding syntax tree. It navigates through the tree representation of the XPath expression, by traversing the Backbone Path. During this traversal, the system identifies PPFs and assigns a schema relation to the last step of each PPF (using the graph representation of the schema). We call the last step of a PPF the *Prominent Step* and the respective relation *Prominent Relation* of the PPF.

The detailed algorithm for gradually building the SQL equivalent of the XPath expression is presented in Section 4.3 and examples are shown in Table 3. The case where we need to assign multiple relations to the last step of a PPF (e.g., if the last step has a wildcard) is addressed in Section 4.4.

4.2 Joining PPFs

Let's suppose that we want to translate the XPath expression '/A[@x=4]//C'. It is obvious that, in addition to 'C', the 'A' relation must also be involved in the SQL statement, since we need to set a restriction on the 'x' column of this relation (x=4). Dewey encoding is used in order to join the two relations in such a way so as to satisfy the '//' axis. In particular, in order for two elements to have an ancestor-descendant relationship, the Dewey vector of the former must be a prefix of the Dewey vector of the later [9], and similar conditions hold for the structural relationships corresponding to the other XPath axes.

We implement the Dewey position of a node as a binary string consisting of one or more components of 3 bytes each. So, if $d(n)$ denotes the Dewey position of node n and k is its level, we have $d(n)=C_1||C_2||...||C_k$, where '||' is the binary string concatenation operator and C_i a component of the dewey vector.
Each component has its first bit equal to zero, thus ranging from 0 up to 7FFFFF (in hex notation). Using this representation, we can use simple *lexicographical* comparisons between the Dewey positions of two nodes in order to perform a structural join over any XPath axis, as shown by the lemmas below.

Let '$\lhd$', '$\rhd$' be the operators for lexicographically 'greater' and 'smaller' respectively. In what follows, we use the hexadecimal notation for Dewey positions.

Lemma 1: Node n_2 is a descendant of node n_1 if and only if

$$d(n_2) \rhd d(n_1) \wedge d(n_2) \lhd d(n_1) \,||\, 'F'$$

Lemma 2: Node n_2 is a following node of n_1 if and only if: $d(n_2) \rhd d(n_1) \,||\, 'F'$

Proofs of the two lemmas can be found in Appendix A. In a similar manner, we can prove we can use lexicographical comparisons over dewey positions to handle all XPath axes. Table 2 (1-6) lists the XPath axes and the respective conditions in SQL, assuming that relations R2 and R1 correspond to two consecutive steps of a path, the second of which having the axis shown in the left column.

Table 2. Axes handled using Dewey encoding

Axis	SQL Condition	
descendant/ descendant-or-self	R2.dewey_pos BETWEEN R1.dewey_pos AND R1.dewey_pos‖'F'	(1)
ancestor/ ancestor-or-self	R1.dewey_pos BETWEEN R2.dewey_pos AND R2.dewey_pos‖'F'	(2)
following	R2.dewey_pos > R1.dewey_pos ‖ 'f'	(3)
following-sibling	R2.dewey_pos > R1.dewey_pos AND R1.par_id = R2.par_id	(4)
preceding	R1.dewey_pos > R2.dewey_pos ‖ 'f'	(5)
preceding-sibling	R1.dewey_pos > R2.dewey_pos AND R1.par_id = R2.par_id	(6)

Notice that parent and child axes can be handled either with Dewey order comparison or with foreign key referencing, with a join between the same two relations in both cases, but on different columns. In particular, the join conditions for these two axes (following the notation of Table 2) are: for child, R2.par_id = R1.id, and for parent, R2.id = R1.par_id. Our algorithm uses the second way, because it is expected to be faster: foreign key and primary key columns, which are integers, are much smaller than dewey_pos columns, which are binary strings of variable length, and moreover equijoins perform generally better than theta-joins on an RDBMS. For examples, see Table 3 (2) in the next Section.

4.3 PPF-based XPath processing Algorithm

Each time a *Primitive Path Fragment* is parsed, the procedure presented in Algorithm 1 is executed to gradually build the SQL equivalent of the XPath expression.

The algorithm adds the name of the prominent relation in the 'from' clause (line 1) and the appropriate restrictions in the 'where' clause of the SQL statement (lines 2-14). The restrictions depend on the type of the PPF and whether it is the first in the backbone path or not. If the last step of the PPF has predicate(s), then one or more sub-selects are created and added in the main SQL statement (lines 15-16).

If it is a forward PPF, the prominent relation is joined with the 'Paths' relation, to which, in turn, a restriction is set filtering the root-to-node path column (lines 2-3) so as to match the regular expression derived from the PPF. If there are one or more consecutive forward PPFs just before the current PPF, then the regular expression includes the entire forward path.

Algorithm 1 SQL Gradual Building per PPF parsing

```
parsePPF(PPF curPPF){
 1  SQLStmt.getFromClause().AddRelation(
      curPPF.getPromintentRelation());
 2  if (curPPF.isForward()){
 3    SQLStmt.JoinWithPaths(curPPF.getProminentRelation(),
        curPPF.getMaxFarwardPath().createRegularExpr());
    }
 4  else if (currentPPF.isBackward(){
 5    SQLStmt.JoinWithPaths(
        curPPF.getPrevPPF().getProminentRelation(),
        curPPF.getBackwardPath().createRegularExpr());
    }
 6  else{
 7    SQLStmt.JoinWithPaths(curPPF.getProminentRelation(),
        "^./" + curPPF.getLastStepName() + "$")
    }
 8  if (PPF.notFirst()){
 9    if (PPF.isSingleStep()&&
        PPF.getLastStep().getAxis() == "parent")
10      SQLStmt.FKStructuralJoin(
          curPPF.getProminentRelation(),
          curPPF.getPrevPPF().getProminentRelation());
```

```
11    else if (PPF.isSingleStep()&&
        PPF.getLastStep().getAxis() == "child")
12      SQLStmt.FKStructuralJoin(
          curPPF.getPrevPPF().getProminentRelation(),
          curPPF.getProminentRelation());
13    else
14      SQLStmt.DeweyStructuralJoin(
          curPPF.getPrevPPF().getProminentRelation(),
          curPPF.getProminentRelation(),
          curPPF.getStructuralRelationship());
    }
15  if (curPPF.getPredicates()!=NULL)
16    SQLStmt.getWhereClause().AddPredicates(
          curPPF.getPredicates());
  }
```

For a backward PPF, the prominent relation of the *previous* PPF is joined with the 'Paths' relation with the restriction that the path column matches the regular expression derived from the path of the current PPF (lines 4-5). For a single-step PPF whose axis is one of the subsequent: *following-sibling*, *following*, *preceding-sibling* or *proceeding*, the prominent relation is joined with the 'Paths' relation, with the restriction that the path column ends with the step's *name test* (lines 6-7). Table 4 shows 2 examples with PPFs that have the *following-sibling* and *preceding* axes.

If the PPF is not the first of the backbone path, then its prominent relation is also joined (structural join, using Dewey encoding) with the prominent relation of the previous PPF (lines 8-14). Particularly, if the current PPF is a multiple-step PPF or a single-step PPF whose axis is not *child* or *parent*, like those shown in Table 3 (1) and (3) (grey parts of SQL statements) and Table 5, then this join occurs over the dewey_pos columns of the two relations (lines 13-14), according to Table 1. Otherwise, if the PPF has only one step with the *child* or *parent* axes, the join is a natural join on the foreign-key reference (lines 9-12), as illustrated in Table 3 (2).

Table 3. Forward (1,2) and Backward (3) PPFs translation examples

XPath	SQL Translation	
/A[@x=3]/B/C//F	select distinct F.id, F.dewey_pos, F.text from A, F, Paths F_Paths where F.path_id = F_paths.id and REGEXP_LIKE(F_Paths.path, '/A/B/C/.*/F') and C.dewey_pos between A.dewey_pos and A.dewey_pos‖'f' and A.x=3 order by F.dewey_pos	(1)
/A[@x=3]/B …	select … from A, B, Paths B_Paths where B.path_id = B_Paths.id and B_paths.path = '/A/B' and B.A_id = A.id and A.x=3 …	(2)
//F/parent::D/ ancestor::B…	select … from F, Paths F_Paths, B, … where F.dewey_pos between B.dewey_pos and B.dewey_pos ‖'f' and F.path_id = F_Paths.id and REGEXP_LIKE(F_Paths.path, '.*/B/.*/D/F')	(3)

Table 4. Translation examples of steps with following-sibling (1) and preceding axes (2)

XPath	SQL Translation	
//D[@x=4]/ following-sibling::E…	select … from …D, E, … where E.dewey_pos > D.dewey_pos and D.C_id = E.C_id and D.x=4 ...	(1)
//D[@x=4]/ preceding::H…	select … from …D, H, … where D.dewey_pos > H.dewey_pos ‖ 'f' and D.x=4 ...	(2)

Logical predicates are handled as follows: We assume that a predicate consists of one or more predicate clauses, combined with logical operators (or, and, not()). Each predicate clause can be a path, a comparison between a path and an atomic value, or a comparison between paths (predicate join-clause). The logical structure of an XPath predicate is translated into a corresponding logical structure in the 'where' clause of the SQL statement (with the same combination of logical operators and parentheses), where each predicate clause is translated into an 'exists()' clause, incorporating a sub-select statement. In what follows, a step of the XPath expression on which a predicate is attached is called a *predicated step*.

If the predicate clause is a (relative) path or a comparison between a path and an atomic value, the respective sub-select statement is created similarly to the main SQL statement for a given backbone path (as in lines 1-14). The difference is that the prominent relation of the first PPF of the path *inside* the predicate clause, (which is included in the 'from' clause of the sub-select statement) is joined appropriately in the outer SQL select statement to the relation corresponding to the predicated step. An example is shown in Table 5 (1).

If a predicate clause consists only of a Backward Simple Path, instead of joining the prominent relation of this path to the relation corresponding to the predicated step, we can once again exploit path id filtering. Particularly, we add an additional restriction to the root-to-node path of the predicated step so as to match the regular expression equivalent of the backward path within the predicate clause. Table 5 (2) shows an example with a predicate which consists of two backward path predicate clauses.

Table 5. Translation examples of XPath expressions containing predicates

Axis	SQL Condition	
/A/B[C/*/F=2]..	select … from B, Paths B_paths, … where B.path_id = B_paths.id and B_paths.path = '/A/B' and exists (select null from F, Paths F_paths where F.path_id = F_paths.id and REGEXP_LIKE(F_Paths.path, '/A/B/C/.*(/)?F') and F.dewey_pos between B.dewey_pos and B.dewey_pos ‖ 'f' and F.text = 2) …	(1)
//F[parent::D or ancestor::G] …	select … from F, Paths F_paths, … where F.path_id = F_paths.id and REGEXP_LIKE(F_Paths.path, '^.*/G/.*(/)?F') or REGEXP_LIKE(F_Paths.path, '^.*/D/F$'') …	(2)

Finally, if the predicate clause is a comparison between two relative paths, the respective sub-select includes all the prominent relations of the PPFs of the first path, joined properly, all the prominent relations of the PPFs of the second path, also joined properly, and an additional theta-join between the relations corresponding to the last PPF of each path.[1]

After all PPFs have been parsed, the SQL statement is completed by adding the 'distinct' SQL keyword before the projection, so as to avoid duplicates in results, and also the 'order by' clause, at the end of the statement, applied on the dewey_pos column of the prominent relation of the last PPF, so that the tuples of the results are retrieved in document order.

4.4 Eliminating SQL Splitting

As we saw in the previous sections, the prominent step of each PPF causes a relation to be added in the final SQL statement. If the prominent step of a PPF corresponds to more than one relation, the SQL statement needs to be split into multiple statements combined by UNION. For example, the XPath expression 'A/B/*[//F]' contains two PPFs: the 'A/B/*' and '//F', the first of which, evaluated over the XML Schema of Figure 1(a), corresponds to two relations. The SQL translation of the expression has two SQL statements, with different FROM clauses. We call this *SQL splitting*.

Prominent steps of PPFs that appear in predicates do not cause SQL splitting. If such a PPF corresponds to multiple relations, then, instead of splitting the entire SQL statement, only the sub-select corresponding to the predicate clause is split into multiple sub-selects, one for each relation, separated with the 'OR' operator. An example is illustrated in Table 6.

Table 6. SQL translation of XPath query which contains a predicate with an ambiguous path

Axis	SQL Condition
/A/B[C/*]…	select … from B, Paths B_paths, … where B.path_id = B_paths.id and B_paths.path = '/A/B' and exists (select null from D, Paths D_paths Where D.path_id = D_paths.id and REGEXP_LIKE(D_Paths.path, '/A/B/C/.*(/)?') and D.dewey_pos between D.dewey_pos and D.dewey_pos‖ 'f') or exists (select null from E, Paths E_paths where E.path_id = E_paths.id and REGEXP_LIKE(E_Paths.path, '/A/B/C/.*(/)?') and E.dewey_pos between E.dewey_pos and E.dewey_pos ‖ 'f') …

SQL splitting is a significant issue for existing schema-aware XPath to SQL translateon techniques [11]. Consider the XPath expression '/A/B[@x=4]/C/*/F'. If we use existing methods for schema-aware SQL translation, we must first find all possible relation sequences corresponding to the path '/A/B/C/*/F', which are A-B-C-D-F and A-B-C-E-F, and then create a separate SQL statement for each such sequence,

[1] The condition of the theta-join is a comparison between the appropriate columns. The SQL comparison operator is derived from the XPath comparison operator of the predicate clause.

where the relations would be joined (natural joins) per consecutive pair. A more advanced algorithm, such as the one presented in [11], detects that elements B can be nested only to elements A, which means that the join between relations A and B is redundant and could be omitted. In contrast, using PPF-based processing, only relations B and F need to be joined, whereas the wildcard is incorporated into an appropriate regular expression filtering on the root-to-node path values, without the need of using two SQL statements.

The combination of root-to-node path filtering, Dewey encoding and schema-aware mapping can reduce the incidence of SQL splitting, and the concomitant problems of multiple query optimization faced in that case by an RDBMS.

4.5 Omitting Unnecessary Root-To-Node Path filtering

Combining schema knowledge with root-to-node path ids gives an optimization opportunity not present in schema-oblivious systems: under certain circumstances path id filtering is redundant and can be omitted. For example, consider the XPath query '/A/B/C/D'. According to the XML Schema (Figure 1), the only possible root-to-node path of elements 'D' coincides with the path of the XPath query. Therefore, there is no need to join relations 'D' and 'Paths'.

We avoid the unnecessary path index lookup (which results in an SQL join) in the following way: After the corresponding graph for an XML Schema has been created, we mark all nodes of the graph that have a unique path towards the root node of the graph with a 'U-P' (Unique Path) tag, all nodes of the graph that have at least one cycle in a path towards the root with a 'I-P' (Infinite Paths) tag and, finally, the remaining graph nodes with a 'F-P' (Finite Paths) tag. 'F-P' nodes are also assigned a list of all possible root-to-node paths. An example is shown in Figure 2.

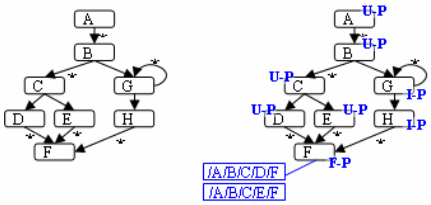


Fig. 2. Marking the XML Schema graph

Relations corresponding to 'U-P' graph nodes are never joined to the Path relation. When an 'F-P' relation is involved in an SQL statement, after translating the PPF path into a regular expression, we test the root-to-node paths of the respective node graph against the regular expression. The relation is joined to the Paths relation and the regular expression restriction is added only if there is at least one such path that doesn't match. Finally, a 'I-P' relation is always joined to the Paths relation with the 'path' column filtered by the regular expression.

5 Experimental Evaluation

In this section we present the results of the experimental testing of the performance of PPF-based processing. We use Oracle 10g (release 10.1 for Windows) as our RDBMS backend. First, we compare PPF-based processing on a schema-aware XML-to-relational system against a schema-oblivious version of same. Moreover, we compare the performance of PPF-based processing with the MonetDB/XQuery [18], which is an XQuery implementation on the MonetDB server backend, the XPath Accelerator mapping scheme [2], which we implemented over Oracle 10g, and a major commercial RDMS with a built-in XML shredding mechanism.

Table 7. The XPath queries used for DBLP XML document

QD1	//inproceedings/title[preceding-sibling::author = 'Harold G. Longbotham']
QD2	/dblp/inproceedings[year>=1994]//sup
QD3	/dblp/inproceedings/title/sup
QD4	//i[parent::*/parent::sub/ancestor::article]
QD5	/dblp/inproceedings[author=/dblp/book/author]/title

We experiment with both synthetic and real data. For synthetic data we use the XMark [20] benchmark variation for XPath, called XPathMark [21]. Using the XMark XML generator, we created two XML documents of 12 and 113 MBs. From the query set of the benchmark we chose a subset of 16 queries, shown in Appendix B, that are compatible with the XPath subset supported by our system. We also added XPath query **Q-A**: /site/open_auctions/open_auction[bidder/date = interval/start] which contains a join predicate clause. We also use the 130MB DBLP XML database.[2] The query set for this database is shown in Table 7.

Experiments were performed on a Pentium 4 PC at 3GHz with 1 GB RAM, running Windows XP. All the queries were executed against a cold cache. For each query we recorded the average time for 5 repetitions.

5.1 Schema-aware vs. Schema oblivious storage

PPF-based processing can be applied both in a schema-aware and a schema-oblivious setting. Moreover, some of the individual techniques we use, notably exploiting Dewey encoding for structural join, have been employed in the context of schema-oblivious systems. We implemented a variation of PPF-based processing tailored to an Edge-like mapping and compared its performance with the PPF-based processing algorithm described in the previous Section. The results confirm our intuition, that apportioning XML content into several relations leads to better query execution performance, and support our decision to focus on implementing and improving PPF-based processing on a schema-aware system (our schema-based optimizations are described in Section 4.5).

[2] Available from http://www.cs.washington.edu/research/xmldatasets/

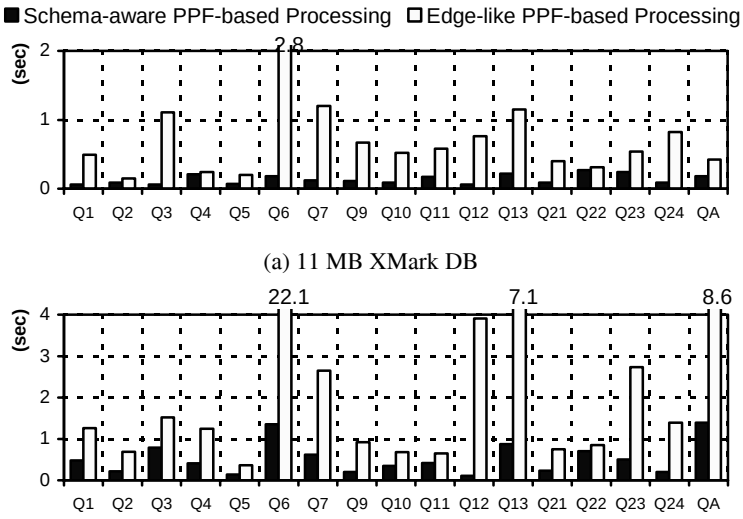


Fig. 3. Schema-aware vs schema-oblivious PPF-based Processing performance

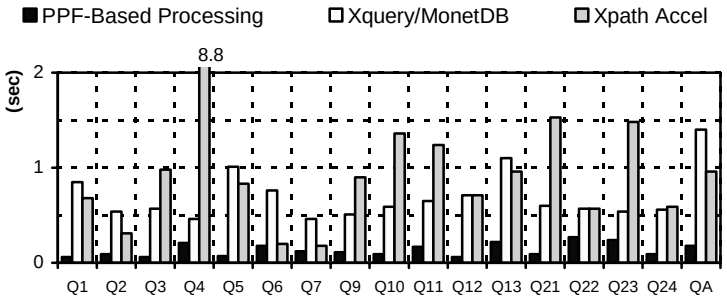
The results of the experiments are shown in figure 3. The most remarkable differences are observed in queries involving structural joins, such as Q6, Q7, Q-A, DQ2 and DQ5. This is due to the fact that, in the schema-oblivious version, these joins are self-joins that join a large relation to itself, in contrast with schema-aware structural joins that join much smaller relations. Even when a concatenated (composite) index is used in the dewey_pos and path_id columns, which is the case, this is larger in the schema-oblivious mapping, compared to all such indices for each mapping relation in the schema-aware mapping, thus the number of I/O is much bigger. Q12 and Q13 also perform remarkably worse in the schema-oblivious version of PPF-based processing. Another factor is that an extra join must take place, since in Edge-like mapping schemes attributes cannot be inlined as columns in the central element relation. Therefore, they are mapped either as separate tuples in the central relation or as tuples in a separate relation exclusively dedicated for attribute storage[3]

5.2 Performance evaluation of PPF-based processing

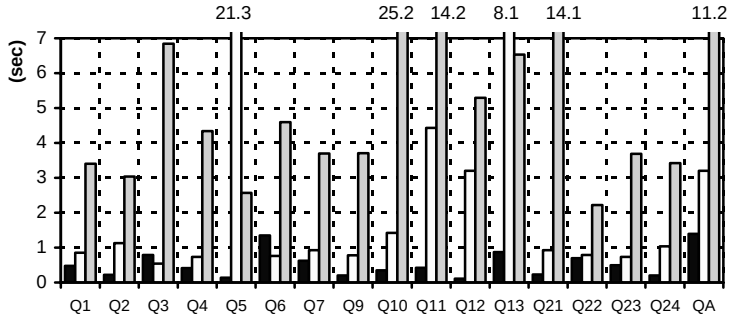
The comparison among PPF-based processing, MonetDB/XQuery and XPath Accelerator scheme is indicative and does not allow us to draw absolute conclusions. We should take into account that the two systems are implemented over different DBMS back-ends, the comparison of which is beyond of the scope of this paper. Moreover, MonetDB/XQuery employs a number of optimizations, most notably the use of staircase joins for structural join. Combining PPF-based processing with join techniques

[3] We used the second option

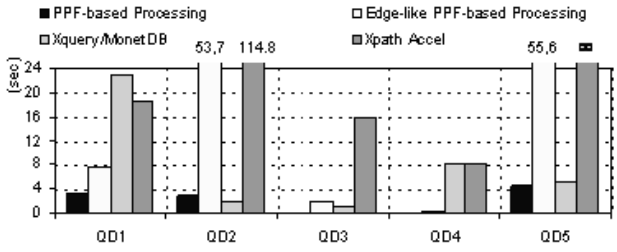
specifically designed for XML data, such as staircase join, is the topic of future work. The comparison between PPF-based processing and our implementation of XPath Accelerator is more direct and allows us to draw more concrete conclusions about the benefits of PPF-based processing. Notice that the translation of the test queries into SQL was made manually following strictly the '*Staked Out Query Window Sizes'* algorithm presented in [2]. As for the commercial RDBMS, the built-in shredding/XPath processing mechanism supports only three of the XPathMark queries, and hence it is not shown in the Figures below (the numbers are available in Appendix C).



(a) 11 MB XMark DB



(b) 113 MB XMark DB



(c) 130 MB DBLP DB

Fig. 4. Comparison of PPF-based processing to other systems/techniques

The two major reasons why PPF-based processing outperforms the other systems in almost all queries are the following:

- the joins performed in PPF-based processing occur between much smaller relations, and
- the number of joins in an average SQL translation is much smaller due to the handling of PPFs using regular expression filtering.

These two factors do not affect all queries. For example, queries Q6, Q7 and QD2 involve structural joins that cannot be removed with root-to-node path filtering. Q6 on the large XMark document and DQ2 are faster in MonetDB possibly because of other optimizations applied by MonetDB, the staircase join being one of these. We will explore combining such optimizations with our techniques as part of future work. Notice especially the performance gains of PPF-based processing on Q5 and QD4. Our technique achieves this level of performance because these queries involve predicate clauses consisting only of backward simple paths, a case which our algorithm handles completely by exploiting path id filtering (see Table 5-2) instead of using structural joins.

6 Related Work

Numerous systems and techniques have been developed in the last few years [4,5] that map XML structures to relations using schema information. Shrex [4] is a system for shredding, loading and querying XML documents using relational systems. The mapping mechanism is flexible, allowing the user to define mapping practices. The XPath-To-XML translation mechanism is rather conventional, since it handles paths with sequential foreign key joins, in contrast to our proposal which involves root-to-node path filtering. Shrex also suffers from the problem of SQL splitting, which we tackle, as described in section 3.4. In [11] an algorithm is presented which, under certain circumstances, alleviates the SQL splitting problem removing at the same time joins which are implied by the schema as redundant. Our proposal alleviates the SQL splitting problem and reduces the number of joins by using PPF-based processing, and uses schema information in order to reduce redundant root-to-node path filtering, as described in Section 3.5. The evaluation of recursive paths is handled in [12], where an algorithm is presented exploiting recursion capabilities of SQL99. In our approach, recursive queries are not considered as a separate problem: a recursive path will be translated into an appropriate regular expression which will be used to detect all matching root-to-node paths.

For schema-oblivious mappings, one of the most comprehensive proposals is XPath Accelerator [2], based on region encoding. XQuery/MonetDB [18] is an XQuery implementation based on the XPath Accelerator on top of the MonetDB DBMS. It supports a large portion of the XQuery recommendation achieving, at the same time, remarkably good performances due to several optimizations and advanced query processing techniques, such as the staircase joins. A detailed comparison of PPF-based encoding to XPath Accelarator and MonetDB/XQuery can be found in Section 5.2. Other Edge-oriented proposals exploit also region encoding, such as [8] and XRel[3]. In [3], region encoding is combined with root-to-node path storage in order to reduce the number of structural joins. Instead of region encoding, [16] uses an update-friendly variation of dewey encoding, called ORDPATH [19], in combina-

tion to root-to-node path storage. However both [3] and [16] support only forward axes and moreover their root-to-node path testing cannot discriminate between wildcards and '//'. In particular, XRel does not handle wildcards, whereas [16] handles '//' with structural join.

7 Conclusions and Future Work

In this paper, we describe a framework based on identifying, processing and combining *Primitive Path Fragments* for processing XPath expressions on a relational back-end. Our technique significantly limits the number of SQL joins required, takes advantage of the strengths of modern SQL query processors and exploits XML schema information to achieve big performance gains with low implementation complexity. Based on our work and the experimental results so far, we can conclude that PPF-based processing is an efficient and easy to implement technique for handling a large XPath subset, including all axes, on top of a relational back end. Root-to-node path indexing is very beneficial for PPF processing when combined with regular expression matching, and is used to holistically process a PPF without any structural joins. We believe that PPF-based processing can be easily adapted to native XML processing systems, and can be combined with native XML join techniques such as twig join [28], yielding performance benefits simply by reducing the number of joins required for a specific XPath expression. We are currently exploring this issue.

Schema-aware mapping can benefit query performance as long as it is combined with proper XML structural encoding techniques, such as presented in this paper. Furthermore, by exploiting XML Schema information, in some cases even root-to-node path filtering is redundant and, thus can be omitted.

Our PPF-based XPath to SQL translation algorithm leads to SQL queries that involve only the necessary relations, with the minimum number of structural joins and the maximum exploitation of root-to-node path ids. Our technique also deals with the problem of SQL splitting, which is common for schema aware mapping systems.

We are currently investigating techniques for increasing the efficiency of XPath processing by exploiting special features of commercial RDBMSs. An interesting question for our technique, explored also in [27] (though with a focus on XML publishing), is how to teach the RDBMS optimizer to produce more efficient query plans.

References

1. D.Florescu, D.Kossmann: Storing and Querying XML Data using an RDMBS. Data Engineering Bulletin, 22(3), 1999.
2. T.Grust, M. V.Keulen, J.Teubner: Accelerating XPath Evaluation in Any RDBMS. ACM Transactions on Database Systems, Vol. 29, No. 1, 2004.
3. M.Yoshikawa, T.Amagasa, T.Shimura, S.Uemura: XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. ACM Transactions on Internet Technology, Vol. 1, No. 1, 2001.
4. S.Amer Yahia, F.Du, J.Freire: A Comprehensive Solution to the XML-to-Relational Mapping Problem. WIDM'04, November 12–13, 2004.

5. J.Shanmugasundaram, K.Tufte, et al.: Relational Databases for Querying XML Documents:Limitations and Opportunities. Proc. of the 25th VLDB Conf., 1999
6. J.Clark, S.DeRose: XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath.
7. S.Boag, D.Chamberlin, et al. : Query 1.0: An XML Query Language. W3C Working Draft 04 April 2005. http://www.w3.org/TR/xquery/.
8. D.DeHaan, D.Toman, M.P.Consens, M. T. Ozsu: A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding, SIGMOD 2003, San Diego.
9. I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E.Shekita, and C. Zhang: Storing and querying ordered XML using a relational database system. SIGMOD Conf., 2002
10. A.Virmani, S.Agarwal, R.Thathoo, S.Suman, S. Sanyal: A Fast XPATH Evaluation Technique with the Facility of Updates. CIKM '03 ACM
11. R.Krishnamurthy, R.Kaushik, J.F.Naughton: Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? Proc. of the 30th VLDB Conf., 2004.
12. R.Krishnamurthy, V.T.Chakaravarthy, R. Kaushik, J.F.Naughton: Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation. Proc. of the 20th ICDE, 2004.
13. A.Berglund, S.Boag, et al.: XML Path Language (XPath) 2.0. W3C Working Draft 2005. http://www.w3.org/TR/xpath20/.
14. G.M.Sur, J.Hammer, J.Siméon, UpdateX - An XQuery-Based Language for Processing Updates in XML. PLAN-X 2004 In. Proc., BRICS Notes Series NS-03-4.
15. K.Deschler, E.Rundensteiner: MASS: A Multi-Axis Storage Structure for Large XML Documents. CIKM '03, 2003, USA.
16. S.Pal, I.Cseri, O.Seeliger, G.Schaller, L.Giakoumakis, V.Zolotov: Indexing XML Data Stored in a Relational Database. Proc. of the 30th VLDB Conference, 2004.
17. IEEE Std 1003.1, Open Group Technical Standard
18. P.Boncz, T.Grust, M.Keulen et al.: PathFinder/MonetDB: XQuery-The Relational Way. Proc. of the 31st VLDB Conference, 2005
19. P. O'Neil, E. O'Neil1, S.Pal, I.Cseri, G.Schalle, N.Westbury: ORDPATHs: Insert-Friendly XML Node Labels. SIGMOD 2004, Paris.
20. Schmidt A., Waas F., Kersten M., et al..: XMark: A Benchmark for XML Data Management. In Proc. of the 28th VLDB Conference, 2002
21. M. Franceschet. XPathMark: an XPath benchmark for the XMark generated data. XSym 2005: 129-143.
22. D. Florescu et al. The BEA streaming XQuery processor. VLDB Journal 13(3), 2004.
23. Y. Papakonstantinou, V. Vassalos. Architecture and Implementation of an XQuery-based Information Integration Platform. IEEE Data Eng. Bull. 25(1): 18-26 (2002)
24. H. Schöning, J. Wäsch. Tamino - An Internet Database System. In Proc. EDBT 2000
25. XML Schema. http://www.w3.org/XML/Schema
26. A.Balmin, Y.Papakonstantinou: Storing and querying XML data using denormalized relational databases. Springer-Verlag 2004.
27. S. Amer-Yahia, Y. Kotidis, D. Srivastava: Teaching Relational Optimizers About XML Processing. XSym 2004: 158-172
28. N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: optimal XML pattern matching. SIGMOD Conference 2002: 310-321
29. T. Fiebig, S. Helmer, et al.: Anatomy of a native XML base management system. VLDB J. 11(4): 292-314 (2002)
30. S. Paparizos, S. Al-Khalifa, et al.: TIMBER: A Native System for Querying XML. SIGMOD Conference 2003: 672
31. Abiteboul, S., Cluet, S., and Milo, T. 1993. Querying and Updating the File. In *Proc. VLDB Conf. 1993*: 73-84.

Appendix A

Lemma 1: Node n_2 is a descendant of node n_1 if and only if

$$d(n_2) \lessdot d(n_1) \wedge d(n_2) \lessdot d(n_1) \,||\, \text{'F'}$$

Sketch of Proof. ***Only if:*** Since n_2 is a descendant of n_1, $d(n_1)$ is lexicographically smaller than $d(n_2)$, so $d(n_2) \lessdot d(n_1)$. We also know that $d(n_1)$ is a prefix of $d(n_2)$, so:

$$d(n_2) = d(n_1) \,||\, S, \text{ where } S = C_{2,k+1} \,||\, C_{2,k+2} \,||\, \ldots \,||\, C_{2,l}. \quad (1)$$

Each Dewey component represents a number from 0 to 7FFFFF, so:

$C_{2,k+1} \lessdot \text{'7FFFFF'} \Rightarrow C_{2,k+1} \lessdot \text{'F'} \Rightarrow C_{2,k+1} \,||\, C_{2,k+2} \,||\, \ldots \,||\, C_{2,l} \lessdot \text{'F'} \overset{(1)}{\Rightarrow} S \lessdot \text{'F'} \Rightarrow d(n_1) \,||\, S \lessdot d(n_1) \,||\, \text{'F'}$. Consequently: $d(n_2) \lessdot d(n_1) \,||\, \text{'F'}$.

If: $d(n_1) \lessdot d(n_2) \lessdot d(n_1) \,||\, \text{'F'} \Rightarrow d(n_1)$ is a prefix of $d(n_2) \Rightarrow n_2$ descendant of n_1. ♦

Lemma 2: Node n_2 is a following node of n_1 if and only if: $d(n_2) \lessdot d(n_1) \,||\, \text{'F'}$

Sketch of Proof. In order for n_2 to be a following node of n_1:

$(d(n_2) \lessdot d(n_1)) \wedge \neg(\text{n2 descendant of n1}) \Leftrightarrow$
$(d(n_2) \lessdot d(n_1)) \wedge \neg (d(n_2) \lessdot d(n_1) \wedge d(n_2) \lessdot d(n_1) \,||\, \text{'F'}) \Leftrightarrow$
$(d(n_2) \lessdot d(n_1)) \wedge (d(n_2) \lessdot d(n_1) \vee d(n_2) \lessdot d(n_1) \,||\, \text{'F'}) \Leftrightarrow$
$d(n_2) \lessdot d(n_1) \wedge d(n_2) \lessdot d(n_1) \,||\, \text{'F'} \Leftrightarrow d(n_2) \lessdot d(n_1) \,||\, \text{'F'}$. ♦

Appendix B: XPathMark queries used in Section 5

Q1	/site/regions/*/item
Q2	/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/ text/keyword
Q3	//keyword
Q4	/descendant-or-self::listitem/descendant-or-self::keyword
Q5	/site/regions/*/item[parent::namerica or parent::samerica]
Q6	//keyword/ancestor::listitem
Q7	//keyword/ancestor-or-self::mail
Q9	/site/open_auctions/open_auction[@id='open_auction0']/bidder/preceding-sibling::bidder
Q10	/site/regions/*/item[@id='item0']/following::item
Q11	/site/open_auctions/open_auction/bidder[personref/@person='person1'] /preceding::bidder[personref/@person='person0']
Q12	//item[@featured='yes']
Q13	//*[@id]
Q21	/site/regions/*/item[@id='item0']/description//keyword/text()
Q22	/site/regions/namerica/item | /site/regions/samerica/item
Q23	/site/people/person[address and (phone or homepage)]
Q24	/site/people/person[not(homepage)]

Appendix C: Aggregate Experimental Results Table

	# of nodes	PPF	Edge-like PPF	XQuery/ MonetDB	Com-mercial RDBMS	XPath Accel.	# of nodes	PPF	Edge-like PPF	XQuery/ MonetDB	Com-mercial RDBMS	XPath Accel.
Q1	2175	0.06	0.49	0.85	N/A	0.68	21750	0.48	1.26	0.85	N/A	3.40
Q2	361	0.09	0.15	0.54	N/A	0.31	4127	0.22	0.69	1.125	N/A	3.04
Q3	7014	0.06	1.11	0.57	N/A	0.98	69969	0.79	1.52	0.54	N/A	6.84
Q4	3514	0.21	0.24	0.46	N/A	8.86	34879	0.41	1.24	0.73	N/A	4.34
Q5	1100	0.07	0.20	1.01	N/A	0.83	11000	0.14	0.36	21.28	N/A	2.57
Q6	2778	0.18	2.8	0.76	N/A	0.20	27878	1.35	22.1	0.76	N/A	4.6
Q7	883	0.12	1.2	0.46	N/A	0.18	8884	0.62	2.65	0.93	N/A	3.70
Q9	3	0.11	0.67	0.51	N/A	0.9	8	0.20	0.92	0.78	N/A	3.71
Q10	2174	0.09	0.52	0.59	N/A	1.36	21749	0.35	0.68	1.42	N/A	25.18
Q11	1	0.17	0.58	0.65	N/A	1.24	0	0.42	0.65	4.43	N/A	14.17
Q12	227	0.06	0.76	0.71	N/A	0.71	2210	0.11	3.91	3.20	N/A	5.29
Q13	6025	0.22	1.15	1.10	N/A	0.96	60250	0.87	7.11	8.17	N/A	6.53
Q21	1	0.09	0.4	0.6	N/A	1.53	1	0.23	0.75	0.93	N/A	14.15
Q22	1100	0.27	0.31	0.57	N/A	0.57	11000	0.70	0.85	0.79	N/A	2.22
Q23	952	0.24	0.54	0.54	0.42	1.48	9506	0.50	2.73	0.73	1.42	3.69
Q24	1304	0.09	0.82	0.56	0.53	0.59	12762	0.20	1.39	1.04	0.32	3.42
QA	8	0.18	0.42	1.40	1.48	0.96	64	1.39	8.67	3.20	3.03	11.2

	# of nodes	PPT	Edge-like PPT	MoneDB /XQuery	XPath Accelerator
QD1	2	3.11	7,6	22,93	18,53
QD2	465	3,09	53,71	1,86	114,88
QD3	577	0,09	1,89	1,18	15,97
QD4	1	0,07	0,16	8,17	8,15
QD5	12178	4,58	55,62	5,18	∞