Tree Logical Classes for Efficient Evaluation of XQuery

Stelios Paparizos*
University of Michigan
spapariz@umich.edu

Yuqing Wu*
University of Michigan
yuwu@umich.edu

Laks V. S. Lakshmanan†
University of British Columbia
laks@cs.ubc.ca

H. V. Jagadish*
University of Michigan
jag@umich.edu

ABSTRACT

XML is widely praised for its flexibility in allowing repeated and missing sub-elements. However, this flexibility makes it challenging to develop a bulk algebra, which typically manipulates sets of objects with identical structure. A set of XML elements, say of type book, may have members that vary greatly in structure, e.g. in the number of author sub-elements. This kind of heterogeneity may permeate the entire document in a recursive fashion: e.g., different authors of the same or different book may in turn greatly vary in structure. Even when the document conforms to a schema, the flexible nature of schemas for XML still allows such significant variations in structure among elements in a collection. Bulk processing of such heterogeneous sets is problematic.

In this paper, we introduce the notion of logical classes (LC) of pattern tree nodes, and generalize the notion of pattern tree matching to handle node logical classes. This abstraction pays off significantly in allowing us to reason with an inherently heterogeneous collection of elements in a uniform, homogeneous way. Based on this, we define a *Tree Logical Class(TLC)* algebra that is capable of handling the heterogeneity arising in XML query processing, while avoiding redundant work. We present an algorithm to obtain a TLC algebra expression from an XQuery statement (for a large fragment of XQuery). We show how to implement the TLC algebra efficiently, introducing the nest-join as an important physical operator for XML query processing. We show that evaluation plans generated using the TLC algebra not only are simpler but also perform better than those generated by competing approaches. TLC is the algebra used in the TIMBER [8] system developed at the University of Michigan.

1. INTRODUCTION

XML is in wide use today, in large part on account of its flexibility in allowing repeated and missing sub-elements. However, this flexibility makes it challenging to develop an XML management system. In this paper, we follow the algebraic native XML management approach and focus on tree algebras like [4, 9] over node algebras [10, 18]. A bulk algebra normally requires manipulation of sets of objects that are structurally homogeneous; but this statement is in contrast with the nature of XML query processing. Tree algebras in the past tried to solve this problem by flattening everything and then having to do extra work (grouping, redundant matches etc.) to produce the correct result.

In this paper, we introduce the notion of a *tree logical class* and show that it is possible to define bulk operations on structurally heterogeneous sets of trees by inducing homogeneity through a *tree logical class reduction*. With this as the basis, we define a Tree Logical Class (TLC) algebra and demonstrate its utility in evaluating XQuery. To describe better the problems that we solve, we take a step back and begin with some background and a motivating example.

1.1 Background

An XML database is often described as a forest of rooted node-labeled trees. A query against such a database is often decomposed into one or more *XML query patterns* (or *twigs*), each of which is also represented as a rooted node-labeled tree. An edge in an XML query pattern represents a structural inclusion relationship, between the elements represented by the respective pattern tree nodes. The inclusion relationship can be specified to be either immediate (parent-child relationship) or of arbitrary depth (ancestor-descendant relationship).

Given an XML database and a query pattern, the witness trees (pattern tree matchings) of the query pattern against the database are a forest such that each witness tree consists of a vector of data nodes from the database, each matches to one pattern tree node in the query pattern, and the relationships between the nodes in the database satisfy the desired structural relationship specified by the edges in the query pattern. The set of witness trees obtained from a pattern tree match are all structurally identical. Thus, in tree algebras [4, 9], a pattern tree match against a variegated input can be used to generate a structurally homogeneous input to an algebraic operator.

1.2 Motivation

Consider a query (against the XMark[13, 21] schema) that looks for the bidders who are older than 25 and are bidding on auction items that have been bidden by at least 5 bidders. For each such bidder and each item he/she bids on, the name of the bidder and the info of all the bidders who bid on the auction item are to be returned. The query (in XQuery [2]) is shown in Figure 1. An algebraic representation of this query is shown in Figure 2, using an algebra similar to TAX[9]. Other algebras will produce plans similar in spirit. There are several concerns with the plan in Figure 2.

Structural Clustering. XQuery semantics frequently requires that nodes be clustered based on the presence of specified structural relationships. The RETURN clause in Q1 requires the complete

*Supported in part by NSF under grant number IIS-0208852.
†Supported in part by grants from NSERC (Canada), NCE/IRIS, and a fellowship from BC Advanced Systems Institute.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2004 , June 13-18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 . . . $5.00.

```
FOR $p IN document("auction.xml")//person
FOR $o IN document("auction.xml")//open_auction
WHERE count($o/bidder) > 5 AND $p/age > 25
      AND $p/@id = $o/bidder//@person
RETURN
<person name={$p/name/text()}> $o/bidder </person>
```

Figure 1: Example Query Q1

subtree rooted at each qualifying $o/bidder node. Similarly, the count function in the WHERE clause of Q1 requires as input the set of all bidders for each open auction.

A traditional pattern tree match returns a set of witness trees satisfying the pattern, for example, two-node trees for each pair of open_auction and bidder. These witness trees must then be grouped based on the parent (or root) node, the open_auction ($o). At this point, aggregate function *count* can be applied to determine the number of bidders in each group. Further structural manipulation may be required to reconstruct the original XML structure. It would be nice if (relevant portions of) the original XML structure could somehow be retained.

An alternative approach for finding structural relatives of a node is to perform tree navigation. This approach could be very costly, especially when the tree is deep and there are "//"s in the path expression. It also does not easily lend itself to bulk processing. We will return to navigational techniques when we get to the experimental evaluation part of this paper - until then, we will say no more about this.

Redundant Accesses. In XQuery, it is not uncommon that the same element tag name is used in multiple places. In query Q1, $o/bidder is used in the value join predicate in the WHERE clause, as the parameter to the aggregate function, and finally, as the root of the subtrees to be returned. In Figure 2 a bidder node appears multiple times in the pattern trees, at least one for each appearance in the query. Each algebraic operator typically performs its own pattern tree match, redoing the same selection on bidder time and time again. This redundant pattern matching work is unavoidable for operators that require a homogeneous set as their input.

Redundant Tree Matching. Intermediate results may lose track of previous pattern matching information and can no longer identify data nodes that match to a specific pattern tree node in an earlier operator. For instance, having identified person/@id matches in the very first step at the bottom left of Figure 2, we have no clue in subsequent steps whether any particular person node has an id attribute. To determine this, the same person/@id sub tree is applied in the input for the succeeding *Projection* and *Selection* operations. This either imposes severe limits on our ability to apply rewriting optimizations or leads to redundant tree matching procedures.

In summary, the repetitive pattern-tree match process is breaking up input trees into small pieces, which are then painstakingly re-assembled through grouping and projection. Not only is this non-intuitive, but it also must have a negative effect on performance. Furthermore, successive operators often match very similar patterns, repeating work unnecessarily. These problems only get worse for complex queries, such as Query Q2, shown in Figure 3.

1.3 Our Contributions

In this paper we develop techniques for bulk manipulation of heterogeneous sets of trees. In particular, we extend the notion of the pattern tree by annotating its edges, and generalizing the semantics of the pattern tree match to produce heterogeneous sets of witness trees. We introduce the concept of logical class for nodes in a tree, and manipulate all members of a logical class in identical fashion. We develop TLC, a new tree algebra that manipulates trees based

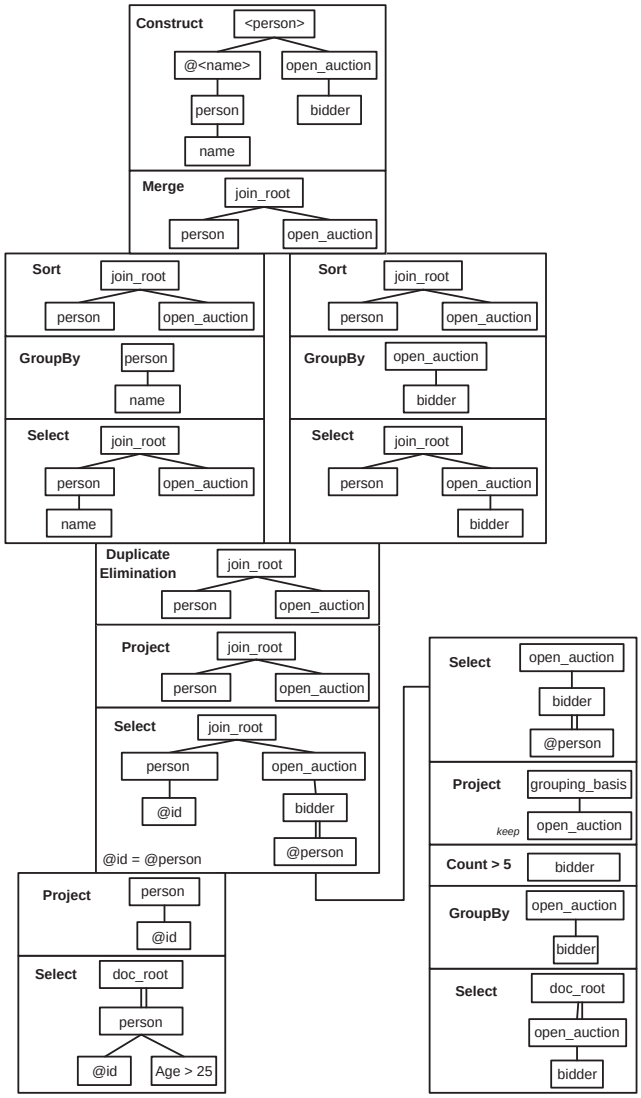

Figure 2: An Algebraic Plan of Example Query Q1. The evaluation is bottom up. For clarity, we have presented this as an evaluation "plan", and pictured the pattern tree used by each operator, rather than writing out a complex formal expression. Many details have been suppressed. The operators used are borrowed from TAX, we define the TLC variants in section 2.3.

on logical class values. These central concepts are all introduced in section 2.

We present an algorithm to obtain TLC algebra representations for a large fragment of XQuery, in section 3. We introduce two additional operators, present rewriting rules to make use of them, and show how to avoid redundant data accesses and redundant pattern matchings with the help of these operators, in section 4. In section 5, we define new physical operators and show how to implement them so that they can efficiently support TLC algebra.

In section 6, we show through extensive experimentation, using TIMBER [11], that TLC produces better evaluation plans than TAX [9], the recently introduced GTP technique [4], and a simple navigational approach. We show that query evaluation performance can improve up to an order of magnitude, using our approach.

2. TLC ALGEBRA

The root of the problems in the examples we saw above lies in the fact that XML data is heterogeneous due to missing and repeated

```
FOR $p IN document("auction.xml")//person
LET $a := FOR $o IN document("auction.xml")//open_auction
        WHERE count($o/bidder)>5
              AND $p/@id = $o/bidder//@person
        RETURN <myauction> {$o/bidder}
           <myquan>{$o/quantity/text()}</myquan>
        </myauction>
WHERE $p/age > 25
        AND EVERY $i IN $a/myquan SATISFIES $i>2
RETURN
<person name={$p/name/text()}>{$a/bidder}</person>
```

Figure 3: Example Query Q2

sub-elements. Most algebraic operators require a means for identifying components of an individual datum (e.g. attribute names or column positions when dealing with sets of tuples). How to identify components of a tree, when a given set of trees may have structural variation? Existing tree algebras use pattern tree match as a basic building block: witness trees that result from matching a given pattern tree must by definition all be homogeneous, with structure identical to the pattern tree. Hence nodes in a witness tree are easy to identify, and operators can reference these to evaluate predicates and perform manipulations as required. This technique, while elegant, has a significant limitation – it requires that witness trees have the same size/structure as the pattern tree, and that each matched pattern tree node must appear once and only once in each witness tree.

To overcome this limitation we extend the notion of a pattern tree by annotating edges to permit heterogeneous match results. Then we introduce logical classes of nodes, tie these to annotated pattern trees, and show how they enable us to access heterogeneous sets of trees as if they were all homogeneous. We conclude this section with a brief description of the operators in a TLC algebra defined on this basis.

2.1 Annotated Pattern Trees

Definition 1. Given a pattern tree $Q = (V, E)$ and an edge $e = (u, v) \in E$, the *matching specification* ($mSpec_e$) associated with e specifies how matches to v are to be obtained for each match of u. The value of $mSpec_e$ can be one of the following:

"-" : one and only one match of v is allowed for each match of u in one witness tree.

"?" : zero or one match of v is allowed for each match of u in one witness tree.

"+" : one or more matches of v are allowed for each match of u in one witness tree.

"*" : zero or more matches of v are allowed for each match of u in one witness tree.

We extend the pattern tree definition to include the concept of matching specification.

Definition 2. An Annotated Pattern Tree (APT) is a tree $Q = (V, E)$ where:

- Associated with each $v \in V$ is P_v, which specifies the predicate for individual node match.
- Associated with each edge $e \in E$ ($e = (u, v)$), is a Rel_e which specifies the desired structural relationship between u and v (the value of Rel_e can be either *parent-child* or *ancestor-descendant*), and $mSpec_e$, the matching specification.

The definition of pattern tree match is extended accordingly:

Definition 3. Given a rooted node-labeled tree $T = (V_T, E_T)$ representing an XML database and an annotated pattern tree $Q = (V_Q, E_Q)$, a match of the annotated pattern tree Q on database T is a one-to-many mapping h such that:

- h maps the root of Q to a singleton set.
- For each pattern tree node $u \in V_Q$, $\forall x \in h(u)$: $P_u(x)$.
- For each pattern tree edge $e = (u, v) \in E_Q$:
  - $\forall y \in h(v)\ \exists x \in h(u) : Rel_e(x, y)$
  - $[if (mSpec_e =$"-" $\vee\ mSpec_e =$"+") then $\forall x \in h(u)$ $\exists y \in h(v) : Rel_e(x, y)$].
  - $[if (mSpec_e =$"-" $\vee\ mSpec_e =$"?") then $\forall x \in h(u)$, $\forall y, z \in h(v) :$ [if $Rel_e(x, y)$ and $Rel_e(x, z)$ then $y = z$]].
  - $[if (mSpec_e =$"?" $\vee\ mSpec_e =$ "*") and $(\exists y \in T : P_v(y) \wedge Rel_e(x, y))$ then $h(v)$ is non-empty].
  - $[if (mSpec_e =$"*" $\vee\ mSpec_e =$"+") then $\forall y \in T :$ $(if\ P_v(y) \wedge Rel_e(x, y))$ then $y \in h(v))]$.

"-" is the default matching specification for all edges in the concept of traditional pattern match. The introduction of matching specification lifts the restriction of the one-to-one relationship between a pattern node and a witness tree node and gives us the freedom to generate heterogeneous sets of witness trees for a given annotated pattern tree. Figure 4 shows the example match for an annotated pattern tree. The figure illustrates how annotated pattern trees address heterogeneity on both dimensions (height and width) using variations of annotated edges. So A_1, A_2 and E_2, E_3 are matched into clustered siblings due to the "+" edges in the APT. On the flip side D_1, D_2 matchings will produce two witness trees for the first input tree (the second tree is let through, although there is no D matching) due to the "?" edge in the APT. Each match $h : Q \rightarrow T$ induces a witness tree that we denote $h(Q)$.

Generalized tree patterns (GTP [4]) and APT both extend the classical notion of tree pattern queries. APTs permit quantified matches via matching specifications, which allow only one or all relatives of a given node to be grouped into one match, an ability that GTPs lack. On the other hand, GTPs allow explicit (existential or universal) quantifiers to qualify matches within the tree specification itself. In the TLC algebra, a subsequent Filter operator can be applied to the result of an APT tree match to obtain the effect of a wide range of quantifiers in the form of aggregation selection predicates. Figure 8, operator box *Filter 10* shows how to do this for universal quantification.

2.2 Logical Classes

Once the pattern tree match has occurred we must have a logical method to access the matched nodes without having to reapply a pattern tree matching or navigate to them. A simple variable binding for every node can do the trick but it is not possible in general; an Annotated Pattern Tree match will generate heterogeneous sets of witness trees. We now have the hard task of identifying which node in each tree we are interested in for the next operation. For example, if we would like to evaluate a predicate on (some attribute of) the "A" node in Figure 4, how can we say precisely which node we mean? For this purpose we introduce the notion of a *Logical Class* that corresponds to a set of nodes in a tree.

Definition 4. Given a witness tree $h(Q)$ produced by the matching of an annotated pattern tree Q in some database; a *Logical Class* (LC) of nodes matching v is defined as $LC(v) = \{h(v) : h(v) \in V_{h(Q)}\}$. The *logical class reduction* of a witness tree $h(Q)$

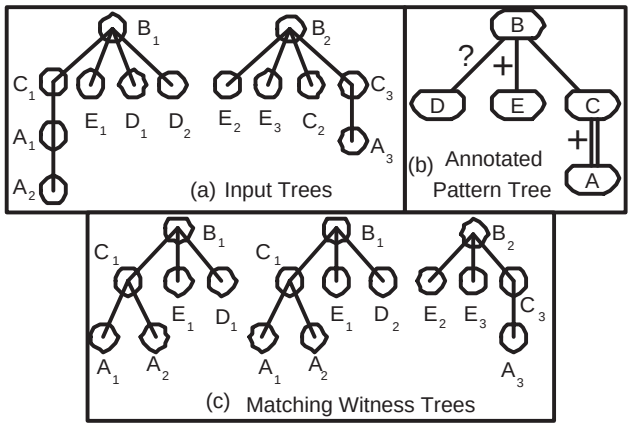

Figure 4: A sample match for an Annotated Pattern Tree. Any edge without an explicit annotation implies the default "-". The double-edged lines represent an ancestor - descendant relationship. Note how the APT addresses heterogeneity on both dimensions (height and width) using variations of annotated edges.

is a tree $LCR(h(Q)) = (V, E)$ isomorphic to the pattern tree Q, defined as follows: (i) Its nodes are logical classes – $V = \{h(v) \mid v$ is a node in $Q\}$; and (ii) it has an edge $(h(u), h(v))$ exactly when the edge (u, v) is present in Q.

Basically, each node in an annotated pattern tree is mapped to a set of matching nodes in each resulting witness tree. Each such set of nodes is called a *logical class*. For example in Figure 4, the E nodes form a logical class for each witness tree. For easy access, we assign a label to each such logical class of nodes. We call this label a *Logical Class Label (LCL)*. The LCL is a unique number associated with each tree. (i.e. a single tree cannot have two LCLs with the same value, e.g. (2), pointing to different LCs). Thus, every node in every tree in any intermediate result is marked as member of at least one logical class. Of course, base data, read directly from the database, has no such logical class association.

Note that the logical class reduction is a tree isomorphic to the pattern tree. Even if there is optionality (via "?" and "*") in the pattern tree, every node will be mapped by h to some (possibly empty) set of nodes. *Thus, even though witness trees resulting from pattern matches can be quite heterogeneous, their logical class reduction is homogeneous.*

We permit predicates on logical class membership as part of an annotated pattern tree specification. This mechanism permits operators late in the plan to reuse pattern tree matches computed earlier. For example, the pattern tree in step 9 of Figure 7 defines a logical class 13 of nodes that have tag name bidder and are immediate children of nodes in the input intermediate result that have been marked as belonging to logical class 5 (which, in this particular query plan, is generated in step 2).

2.3 Algebraic Operators

Every operator in our algebra maps one or more sets of trees[1] to one set of trees. The trees in a set may be heterogeneous. The introduction of LC enables the operators in our algebra to refer to particular nodes in trees; essentially accessing them as logical class reduction trees. When no logical class information exists in a tree (e.g. base data), we assume the class maps to the empty set.

Since a logical class points to a set of nodes in each tree, operators have to deal with sets of elements identified. Some operators can use sets of arbitrary length, while others require that the logical class comprise a singleton set of nodes in each tree, else they generate an error. We discuss this requirement in each operator's description.

Once the above concepts are in place, the details of the operators are straightforward. We present a brief summary of a few popular operators below – most of these are loosely modeled after the operators in TAX [9]:

Filter $F[LCL_f, p, m](S)$: Given an input set of trees S, a filter predicate p, a mode m and a label to a logical class LCL_f: output only the trees in S that satisfy the predicate p for the nodes bound to LC_f. The mode m parameter is used to identify how to iterate over the set of nodes bound to LC_f. The default for m is universal quantification (Every (E))[2] – any predicate evaluated must be true for all nodes in the class, and the result of any manipulations will output all nodes in the class. Existential quantification is covered using at least one (ALO) node in the class. Another possibility is for a predicate to be satisfied at exactly one (EX) node in the logical class (although the class maps to multiple nodes). Other interpretations are also possible, e.g. apply to first element (on the basis of input data node ordering) in the logical class, etc.

Join $J[apt, p](S_l, S_r)$: The operator accepts two input sets of trees S_l and S_r, an Annotated Pattern Tree apt and a predicate p. p uses labels for LCs to describe nodes from trees in S_l and S_r along with their desired content relationship. Each LC that participates in p must map to a singleton set for each input tree. The input apt describes how to generate the result structure using an artificial node and the roots of the logical class reduction for the input trees in S_l and S_r. The edge annotation is always "-" for the left side and can be any of the four $mSpec$ options on the right side. The operator uses the predicate information p to identify matching trees in S_l and S_r and stitches together one input tree in S_l with one or more matching input trees in S_r (according to the right side edge annotation in the apt) .

Select $S[apt](S)$: Select accepts as input a set of trees S and an Annotated Pattern Tree apt. For each tree in the input set S the operator performs a pattern tree match using apt. The output is the entire set of the matching witness trees for all input trees.

Project $P[nl](S)$: The operator accepts as input a set of trees S and a list nl using LC labels to identify sets of nodes (note that the LCs point to arbitrary length sets, including the empty set). For each input tree in S, only the nodes identified by nl are retained in the output; if the output is not a tree, the input tree root is also retained.

Duplicate-Elimination $DE[nl, ci](S)$: Eliminates duplicates of the input set of trees S based on the list of specified nodes nl. nl uses LC labels to identify nodes; each LC in nl must map to a singleton set of nodes. The parameter ci specifies whether to eliminate duplicates based on node content or identifier.

Aggregate-Function $AF[fname, LCL_a, newLCL](S)$: The operator accepts one input set of trees S, the aggregate function name $fname$ (count, max etc.), an LC label LCL_a to describe which nodes to apply the function on and an LC label $newLCL$ for the new node that will be created to hold the result. The function $fname$ is applied on each input tree of S for all nodes mapped to LC_a. LC_a maps to a set of nodes with arbitrary length. Each input tree generates one output tree with the result node added as a sibling of the nodes specified by LC_a; $newLCL$ is the LC label of the new node. Please note that if LC_a maps to the empty set, then the generated node will contain 0 for count and the flag "empty" for all other functions.

[1]The input can be a single tree database.

[2]Please note that the semantics for Every (E) will let the input tree pass if LC_f maps to the empty set.

```
<FLWOR> ::= <ForLetClause> <WhereClause>?  <ORetClause>
<ForLetClause> ::= (<ForClause>|<LetCLause>)+
<ForClause> ::= 'FOR' $var 'IN' <SP> | <FLWOR>
<LetClause> ::= 'LET' $var ':=' <SP> | <FLWOR>
<SP> ::= Simple Path with no branching predicates
<WhereClause> ::= 'WHERE' <WhereExpr>
<WhereExpr> ::= <SimplePredicateExpr> | <AggrPredExpr>
| <ValueJoin> | <QuantifierExpr> | <ANDExp> | <ORExp>
<SimplePredicateExpr> ::= <SP> <Predicate> <Value>
<Predicate> ::= '=', '>', '<', ...
<Value> ::= string or number
<AggrPredExpr> ::= <Aggr(<SP>)> <Predicate> <Value>
<Aggr(<SP>)> ::= 'count('<SP>')' | 'avg('<SP>')' ...
<ValueJoin> ::= <SP> <Predicate> <SP>
<QuantifierExpr> ::= 'EVERY' | 'SOME' $var 'IN' <SP>
'SATISFIES' <SimplePredicateExpr>
<ANDExp> ::= <WhereExpr> 'AND' <WhereExpr>
<ORExp> ::= <WhereExpr> 'OR' <WhereExpr>
<ORetClause> ::= <OrderClause>?  <ReturnClause>
<OrderClause> ::= 'ORDER BY' <SP>1, ..., <SP>n <Mode>
<Mode> ::= 'Ascending' | 'Descending'
<ReturnClause> ::= 'RETURN' <ReturnExpr>
<ReturnExpr> ::= <SP> | <FLWOR> | <Aggr(<SP>)>
| ('<'tag (tag '=' <SP>)*'>' <ReturnExpr> '<'/'tag'>')
```

Figure 5: Grammar for XQuery Fragment.

Construct $C[c](S)$: The construct operator takes a collection of trees S and an annotated construct-pattern tree c as input. An annotated construct-pattern tree is an annotated pattern tree(APT), except it allows facilities for tagging, renaming, and arbitrary tree assembly. Figure 7 shows an example in box *Construct 10*. It specifies that each output tree should have a root with tag person with an attribute name, with value specified by a reference to class (12), and a child element created from the reference to class (13). In our experience, we have found construct to be a low cost operator with few opportunities for optimization through bulk processing.

3. XQUERY TO TLC EXPRESSIONS

Figure 5 shows a simplified, yet substantially expressive, fragment of XQuery that can be translated to a TLC algebra expression. In this section we present the sketch of an algorithm that performs this translation.

To remove complexity of the presented algorithm we simplify the supported path expressions. In XQuery, an XPath expression with branching predicate can always be replaced by a simple XPath expression in the context of a FLWOR query. Hence we operate on XPath expressions without branching predicates, we call such a structure a *Simple Path (SP)*. For example, the XPath expression /bidder/name is a SP, but the XPath expression /bidder[age>25] /name is not.

Pseudocode for the algorithm is shown in Figure 6. Here we will try to convey the intuition behind the algorithm, using as an example the generation of the TLC algebra tree of Figure 7 for Query Q1.

We begin by parsing the two FOR Clauses; they will create two *Selects* and a *Cartesian-Product* of the selection results, corresponding to boxes 1,2 and 5 respectively in Figure 7. *Select 1* only has doc_root//person, *Select 2* only doc_root//open_auction and *Join 5* has no join condition at this stage: the rest of the information shown in the figure will be added by succeeding steps.

The simple predicate expression case is used to add age>25 node with LCL=10 in *Select 1* in Figure 7, and the aggregate predicate expression rule creates the bidder node with LCL=6 in *Select 2* and adds the operators *Aggregate 3* and *Filter 4*.

The value join case will first process the necessary path expressions, adding the @id node with LCL=7 in *Select 1*, the bidder// @person path with LCLs 8 and 9 in *Select 2*. Then it will update the join specification in *Join 5* to include (7) = (9) (@id = @person).

```
Algorithm TLC
Input: a FLWOR expression Output: a TLC algebra plan
Globals LCLCounter VARIABLES OPERATORS PATTERNTREES

procedure SingleBlock(in FLWOR) {
Parse FLWOR, create Reductions for each Grammar Rule
For each Reduction do {
// Processing FOR - LET
Case ForClause ::= FOR $var IN SP
  Set mSpec = "-"
  aptO = SPtoAPT(SP, mSpec)
  add $var to VARS, point to aptO.leafnode
  if (aptO.root is document())
    If empty(OPERATORS) add Select[aptO]
    else if exists(Select[aptA])
      new Join[Cartesian](Select[aptA], Select[aptO])
  else if (aptO.root is $varA)
    addToAPT($varA, aptO, mSpec)
Case LetClause ::= LET $var := SP
  Set mSpec = "*"
  Same with FOR and omitted due to space restrictions
// Processing WHERE
Case SimplePredicateExpr ::= SP Predicate Value
  aptS = SPtoAPT(SP, "-")
  aptS.leafnode.add(Predicate(Value))
  if (aptS.root == $var1) addToAPT($var1, aptS, "-")
Case AggrPredExpr ::= Aggr(SP) Predicate Value
  aptS = SPtoAPT(SP, "*")
  LCLF = aptS.leafnode, newLCL = LCLCounter++
  if (aptS.root == $var1) addToAPT($var1, aptS, "*")
  new AggregateFunction AF[Aggr, LCLF, newLCL]
  new Filter F[newLCL, Predicate(Value), ALO]
  find Select SV that uses apt_V,
    set F.setChild(AF) and SV.parent.setChild(F) and SV.setParent(AF)
Case ValueJoin ::= SPL Predicate SPR
  aptL = SPtoAPT(SPL,"-"), aptR = SPtoAPT(SPR,"-")
  LCLJL = aptL.leafNode, LCLJR = aptR.leafNode
  addToAPT(aptL.root, aptL, "-"), addToAPT(aptR.root, aptR, "-")
  Find join ancestor, add(Predicate(LCLJL, LCLJR))
Case Every ::= 'EVERY' $var 'IN' SP 'SATISFIES' SimplePredExpr
  aptS = SPtoAPT(SP, "*")
  addToAPT(aptS.root, aptS, "*")
  add $var to VARS point to aptS.leafnode
  Process SimplePredExpr, merge with aptS but use "-" edges
  Create Filter operator from SimplePredExpr, use EVERY mode
Case Some ::= 'SOME' $var 'IN' SP 'SATISFIES' SimplePredExpr
  Same with EVERY details omitted, only final Filter uses ALO mode
Case ANDExp ::= WhereExpr 'AND' WhereExpr
  Process left and right input merging the edges
Case ORExp ::= WhereExpr 'OR' WhereExpr
  OR is translated to UNION of the operators produced both sides
  Make sure the root node for each path on both sides is assigned
  the same LCL, even if the tagname is different.
// Processing ORDERBY-RETURN
Case OrderClause ::= ORDER BY SP1, ..., SPn Mode
  for all SPi create aptSi = SPtoAPT(SPi, "-")
  for all aptSi Create a Select S[ aptSi ], add to OPERATORS
  Create Sort[LCL1, ..., LCLn, Mode], add to OPERATORS
Case ReturnClause ::= RETURN ReturnExpr
  Create Project: Keep all bound variables including root if there was a join
  Create NodeIDDE, only on FOR variables
  Parse RE, Create a tree with tags, SP and Aggregates
  Convert each SP to APT, create a Select for that APT
    Replace SP with the LCL referencing the leaf node in APT
  Similarly replace Aggregates(SP), but also add Aggregate[] after Select
  add Construct[parse tree] to OPERATORS
}}

APT function SPtoAPT(SP, mSpec)
  Return an APT from the SP, use Rel from StepAxis, use mSpec for all edges
function addToAPT(VAR, APT, mSpec)
  From VAR retrieve Pattern Tree P and LCL info
    add APT to P as child of LCL using mSpec

procedure NestedQuery(in FLWOR)
  Process FLWOR, if (Nested) then Call SingleBlock for "inner" and "outer"
  Add a join between the outer and inner plan
    Use edge "-" for FOR, edge "*" for LET and RETURN
  Modify plans, join values should pass projects used.
  Also, if inner construct elements are referenced in the
  outer clause then they should survive the outer projection.
```

Figure 6: Algorithm TLC

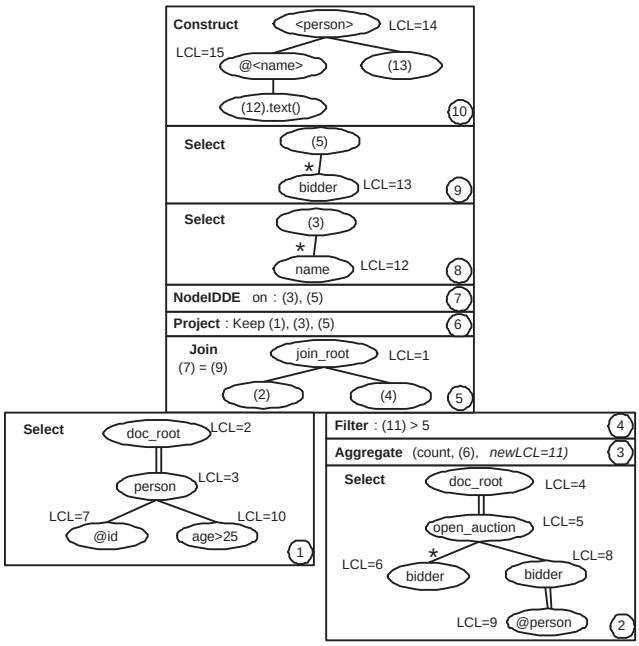

Figure 7: Algebraic Tree for Query Q1. Order of evaluation is bottom-up. Each box represents an operator. Each APT edge has the matching specification marked next to it, except for the default "-" which is omitted for clarity. (ad) relationships use double edges. The LCLs assigned to each class are shown next to the corresponding APT node.

The next several cases in the algorithm do not apply to Q1 (except for the trivial ANDExp case, which was already implicitly folded into our discussion above). We now reach the RETURN clause: at this stage all bound variables are known and all selection predicates have been applied. Before we start processing the RETURN we generate the *Project 6* operator having as parameters the LCLs for the bound variables (LCL=3, LCL=5 in Figure 7) and the root of the input tree if a join was applied (LCL=1). Following the projection we add a node id[3] duplicate elimination *NodeIDDE 7* operator on node ids for only FOR (not LET) bound variables (LCL=3, LCL=5 in Figure 7). Finally, we process the RETURN arguments, adding operators *Select 8*, *Select 9* and *Construct 10*. Note how the selections use pattern tree extensions. Also note how the construct tree is similar to a modified parse tree having substituted the SP in the RETURN clause with LCL references.

Nested FLWOR. We show the full TLC algebra plan for the nested FLWOR query Q2 in Figure 8. Operators 1 and 9-14 are obtained from the outer query in the same manner as Q1. There are no surprises here. Notice how easily universal quantification is handled in *Filter 10*, processing the Every case. Operators 2-8, in the right sub-tree, correspond to the nested (inner) query. Again, most of this is just as before, except that the join condition in the where clause involves a variable bound in the outer query and so is deferred until *Join 9*. In consequence, it becomes necessary to make sure that the nodes referenced for the join will survive the project, construct etc. in operators 5-8. Notice how LCL=9 is added to the projection list in *Project 5* and the construct pattern in *Construct 8*, so that it can participate in *Join 9*. Also since the outer construct references elements of the inner construct they should also be added to the outer projection. Hence (12) is added to *Project 11* so that it can be referenced in the final construct.

[3]This is a very cheap operation, all identifiers are already in memory.

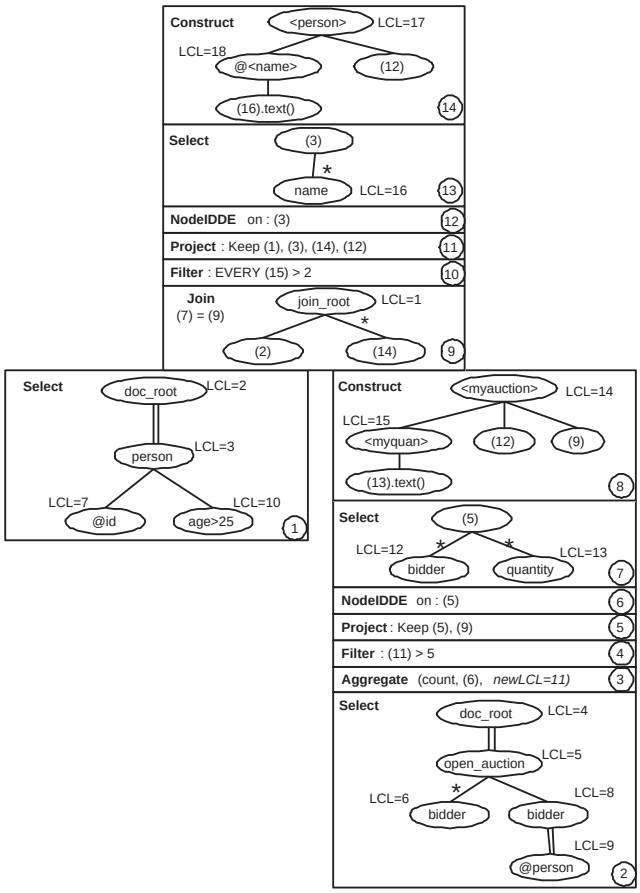

Figure 8: Algebraic Tree for nested Query Q2.

4. ELIMINATING REDUNDANCY

Redundant access and redundant tree matching are drawbacks of traditional tree algebraic representation of XML queries, as we pointed out in Sec. 1.2. The TLC algorithm uses the LC concept to enable pattern tree reuse. In this section we also show how to further improve performance by introducing three new operators (Flatten and Shadow / Illuminate) and the corresponding rewrite rules.

4.1 Pattern Tree Reuse

Given a heterogeneous set of trees, we use pattern tree matches to identify nodes of interest. In an algebraic expression, it is frequently the case that multiple operators use exactly the same pattern tree. It is computationally profligate to re-evaluate the pattern tree matching multiple times, once for each such operator. Instead, we permit the results of a pattern tree evaluation (including the logical class mappings) to persist, and be shared by many of the subsequent operators. Pattern tree reuse is akin to common subexpression elimination. Sometimes, subsequent operators may not use the exact same pattern tree, but rather may use a variation of it. In TLC we can apply additional conditions to the witness trees known to satisfy the original pattern tree match, as well as extend each tree to include new branches. We refer to already matched nodes via the usage of LC labels. For example *Selections* 8 and 9 in Figure 7 use extended pattern trees.

4.2 Flattening

Often nodes with the same tag name in an annotated pattern tree are marked with different predicates and annotated edges. This is

because such nodes appear in multiple places in the query. For example, in the plan for Query Q1 as shown in Figure 7, *Selection 2* has two occurrences of bidder in its associated pattern tree. The bidder node with *LCL=6* is required for the aggregate function in step 3; the one with *LCL=8* is needed, through its descendant person node, for the value join in step 5. This pattern tree will lead to the query evaluator physically accessing each bidder node twice. Reusing the already accessed bidder information is our goal.

The problem is that the two bidder nodes in the annotated pattern tree, while both children of the same open_auction node, have different annotations on their parent edges ("*" versus "−"). One annotation generates witness trees where all bidder children of each open_auction are retained in a single tree, whereas the other annotation requires a separate witness tree for each bidder, open_auction pair. It is frequently the case that we require the first sort of output because of an aggregate (or some other set) computation. In that case, it may be appropriate to evaluate only the former, compute the required aggregates, and then break the trees apart to produce the structure required by the latter, *without accessing the database twice for this purpose.* We introduce the *Flatten* operator to perform this "breaking apart".

Definition 5. **Flatten:** $FL[LCL_P, LCL_C](S)$ takes as input a set of trees S and a pair of LC labels LCL_P, LCL_C pointing to P, C. Class P must bind to a singleton set for each tree and C must map to a set of children of P in each tree. The output sequence of trees is generated in the following way: For every tree T in S { for every node p bound to P and c bound to C in T, produce an output tree T' that is identical to T except we retain only c and drop all other elements in C including their descendant subtrees}.

Flatten operator removes the nested part P-C for each input tree and generates trees in which the matchings to P and C are singleton sets. Figure 9 shows an example of the operator. The input tree Figure 9.a is transformed into two trees in Figure 9.b by $FL[B, E](a)$, then into four trees in Figure 9.c by $FL[B, A](b)$.

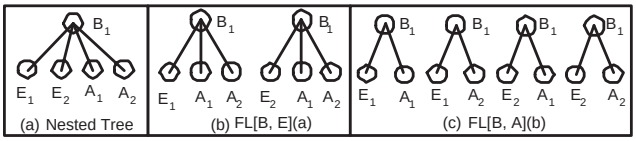

Figure 9: Flatten Operator

Flatten Rewrite Rule.

PHASE 1: (Detect if FL rewrite can be used). Find a Selection $S[apt_A]()$ such that A, B, C are three LCs in the APT apt_A, and $tree(B)$, $tree(C)$ are the subtrees rooted at B, C in apt_A. For the rewrite to be applicable the following conditions must all be satisfied: (i) $\{[(A, B) \in E_{apt_A}] \wedge [(A, C) \in E_{apt_A}] \wedge [mSpec(A, B) = (+ or *)] \wedge [mSpec(A, C) = (- or ?)] \wedge [tree(B) \subseteq tree(C)]\}$ = TRUE, (ii) Let use[$tree(B)$] be some arbitrary operator using B and notuse[$tree(B)$] mean that no other operator accesses B after that step. Check if the operator tree is (1): notuse[$tree(B)$](use[$tree(C)$](use[$tree(B)$]($S[apt_A](\ldots)$))).

PHASE 2: (Perform the rewrite). Let apt_B = apt_A - $tree(C)$ and $apt_D1, \ldots, apt_Dn$ = $tree(C)$ - $tree(B)$ (forest). The operator tree (1) now becomes (2): use[$tree(B)$]($S[apt_D1]$, ..., $S[apt_Dn]$($FL[A, B]$(use[$tree(B)$]($S[apt_B](\ldots)$))).

Figure 10 shows the result of the Flatten rewrite for Example Query Q1, the nodes that qualify for the rewrite are LCL=5 is A, LCL=6 is B and LCL=8 is C in *Selection 2.* To save space, we show only the affected part of the rewritten TLC plan. This example is indicative of the rewrite rule used. Using this rewrite we have avoided going to the database twice and performing two pattern matches to access the same bidder node.

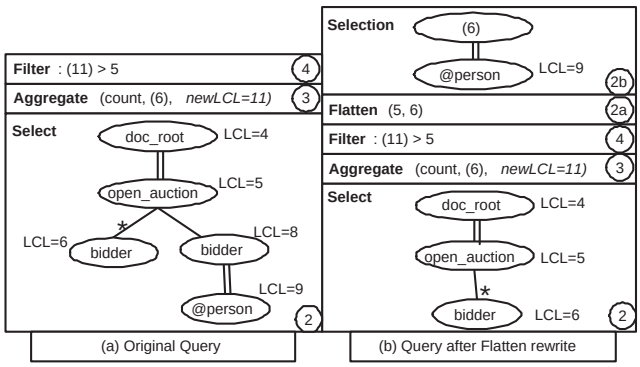

Figure 10: Flatten Rewrite for Query Q1. Please note how Selection 2 from the original plan has been replaced by Selection 2 with a sub-tree of the original pattern tree containing only the path with "*". Flatten 2a has been added after the aggregate has used the bidder nodes. Then selection 2b extends the bidder elements with person. The right block substitutes the left block eliminating redundant tree matches.

4.3 Shadow/Illuminate

When redundant nodes (branches) are to be eliminated from APT, a certain ordering is required between the references to the redundant nodes in order to perform flatten rewrite. In the example above, The aggregate function has to be calculated before the join operations is performed. But the opposite scenario is quite common; suppose we want (or need) to perform some filtering or join operation first and then reconstruct the nested elements. For query Q1, we have a join that uses the bidder path and then require the bidder elements to be clustered together for the result. Currently we would have to do redundant matches to get the bidder elements as seen in Figure 7. In this scenario, the flatten operator could not be used to our advantage. Furthermore, it is not enough to define an operator that is the inverse of flatten and can stitch together multiple intermediate result trees. The reason is that we may wish to include all bidder children of selected open_auction nodes in the result, and not just the bidder children that successfully participated in satisfying some predicate (the join for Q1). To deal with this situation, we introduce the concept of a *shadowed* node, along with two new operators, Shadow and Illuminate. A shadowed node remains a member of its logical class, but is not visible to any operator other than illuminate. In effect, shadowing provides us with a logical means to retain nodes in an intermediate result tree but have them not participating in any operation.

Definition 6. **Shadow** $SH[P, C](S)$: takes as input a set of trees S and a pair of LC labels (P, C). P must bind to a singleton set for each tree and C must map to a set of children of P in each tree. The output sequence of trees is generated in the following way: For every tree T in S { for every pair of nodes p bound to P and c bound to C in T, produce an output tree T' identical to T except all nodes in C except c, including their subtrees, are marked *shadowed*}.

Shadow behaves similarly to Flatten with the extra caching feature of retaining shadowed nodes. The difference between the two can be seen in Figure 11. But simply shadowing the nodes does not help us; we must have a way to access them again when we need to. We use the Illuminate operator for this purpose.

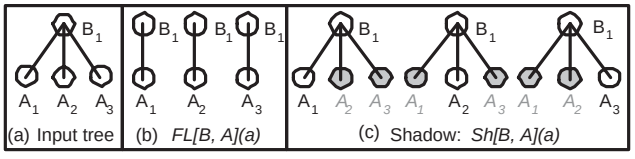

Figure 11: Comparing operators: Flatten vs. Shadow

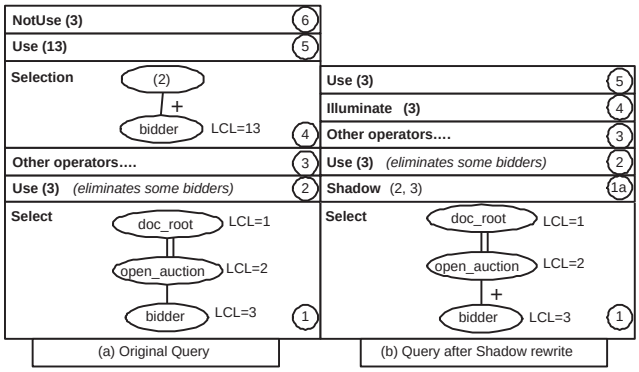

Figure 12: Shadow/Illuminate rewriting procedure. This generic example is indicative of how the procedure is performed. Note how the APT for Selection 1 has been replaced by an APT using an "+" annotated edge followed by the Shadow 1a operator. Selection 4 is replaced by Illuminate 4. We have eliminated redundant access to the database.

Definition 7. **Illuminate** $IL[LCL_i](S)$ takes as input a set of trees S and an LC label LCL_i; for each tree in the S all (inactive) nodes in the indicated logical class LCL_i are rendered active (including their subtrees).

Note that Shadow and Illuminate do not complement each other; Shadow breaks the tree into many trees, but Illuminate does not affect the number of trees at all. Illuminate is necessary because no other operator can access shadowed nodes.

Shadow / Illuminate Rewrite Rule.

PHASE 1: (Detect if `SH/IL` rewrite can be used). Find a Selection $S[apt_A]()$ and a succeeding Selection $S[apt_S]$, such that A, B are two LCs in the apt_A and C is an LC in apt_S, and $tree(B)$, $tree(C)$ are the subtrees rooted at B, C in apt_A, apt_S respectively. For the rewrite to be applicable the following conditions must all be satisfied: (i) $\{[(A,B) \in E_{apt_A}] \wedge [(LCL_A, C) \in E_{apt_S}] \wedge [mSpec(A,B) = (-\ or\ ?)] \wedge [mSpec(LCL_A, C) = (+\ or\ *)] \wedge [tree(B) \subseteq tree(C)]\}$ = TRUE, (ii) Let $\texttt{use}[tree(B)]$ be some arbitrary operator using B and $\texttt{notuse}[tree(B)]$ mean that no other operator accesses B after that step. Check if the operator tree is (1): $\texttt{notuse}[tree(B)](\texttt{use}[tree(C)](S[apt_S](\texttt{use}[tree(B)](S[apt_A](\ldots))))))$.

PHASE 2: (Perform the rewrite). Let apt_B = apt_A but with $mSpec(A,B) = [mSpec(LCL_A, C)$ and $apt_D1, \ldots, apt_Dn$ = apt_S - $tree(C)$ (forest). The operator tree (1) now becomes (2): $\texttt{use}[tree(B)](S[apt_D1], \ldots, S[apt_Dn](\ IL[B](\texttt{use}[tree(B)](SH[A,B](S[apt_B](\ldots)))))))$.

Note that possibly A, B, C can be part of the same apt_A as in the Flatten rewrite. Flatten in that case forces the predicate on the node associated with the (* or +) edge to be evaluated first, i.e. the aggregate. We can use Shadow/Illuminate to perform the opposite, evaluate the predicate on the node associated with (- or ?) edge first, i.e. the join. The details of this rule can be derived from the presented one and are omitted due to space restrictions.

Figure 12 shows the rewrite procedure using a generic TLC plan illustrating how the rules work. Using this rewrite we have avoided going to the database twice and performing two pattern matches to access the same bidder node. The rewrite for Query Q1 can be performed by replacing *Selection 9* with an Illuminate operator and using Shadow in place of Flatten as in Figure 10.

5. PHYSICAL IMPLEMENTATION

We implemented the TLC algebra as part of the TIMBER [8] system developed at the University of Michigan. TIMBER follows the model of a native XML database similar to Tamino [14], X-Hive [20], etc. While most of the implementation details are not within the context of this paper, there are a couple of crucial issues we felt it was important to find the space to describe. These issues involve node identification, pattern tree matching and efficiency during the evaluation of a query.

5.1 Node Identifiers

A node is the minimum storage unit in an XML implementation. We need *Node identifiers* to identify nodes in the database or in memory. We have identified a set of properties that node identifiers must satisfy, they are seen in Figure 13.

1. Provide unique identifiers, *for correctness*.
2. Identify structural relationship between nodes, *for structural joins*.
3. Indicate absolute node order within a tree, *for document ordering (a document is a tree)*.
4. Indicate node order within all nodes of the same class, *looser more flexible ordering notion*.

Figure 13: Important Properties for Node Identifiers

Property 1 is necessary to guarantee uniqueness of a node when we access it; repeated elements in the same level of a tree require a unique method to access each one of them. Property 2 enables us to use the very popular structural join algorithms [1, 3] to perform the pattern tree matches. Properties 3 and 4 are non-intuitive and we will discuss them in further detail.

XML queries must maintain document ordering. An implementation is forced to do all joins using a nested loops algorithm; else document order is no longer maintained. We wanted to overcome this difficulty by requiring our identifiers to indicate absolute tree order (Property 3). Now we can assign node ids based on the document order of each node. If element A precedes B, then node A will have a smaller node id[4]. This technique allows us to sort any sequence of trees based on the node id of the root and re-establish document order. Sorting on node id is usually not too expensive, since all the information needed is typically already in memory. Hence for our joins we use a *sort-merge-sort* algorithm. We sort the two input sequences based on their join values, merge them and then sort the output based on the node id of the first sequence. This way we achieve better performance and linear scalability without sacrificing document ordering.

After implementing our algorithm, we quickly realized that we use many temporary nodes generated during query execution. Such nodes include join_root, group_root, tag nodes generated for the return and so on. To assign identifiers for these nodes, we would have to renumber the in-memory trees, this procedure is described in "Dynamic-Intervals" [6]. Renumbering the in-memory trees is inefficient. But we realized that temporary nodes do not need to satisfy all the node identifier properties. Using logical classes and in-memory tree structures we can identify child inclusion of a temporary node; hence property 2 can be ignored. Although document ordering is important in XML, the temporary nodes are not part of the original document; hence they do not need to satisfy full tree ordering. Instead all nodes of the same class across a sequence of trees, must be sortable[5] (can be sorted using their identifiers). Hence we introduce Property 4 which is a subset of Property 3. So temporary node identifiers must satisfy properties 1 and 4 but do not need to satisfy properties 2 and 3. This principle enables our

[4]The same holds for element A containing B

[5]Necessary for *sort-merge-sort* and output that reflects correct order.

implementation to assign identifiers without having to renumber the entire tree.

5.2 Pattern Tree Matching

A pattern tree match is implemented using a combination of value joins (implemented as in relational systems) and structural joins (implemented as described in [1, 3, 5]). For annotated pattern trees, these join techniques do not suffice. An edge annotated with a "+" or "*" requires the produced witness trees to have "nested" nodes. One simple solution is to treat all edges in an APT as if they were flat ("-" or "?") and then use an explicit grouping procedure combined with a projection to produce the necessary structures. But grouping is an expensive operator, and this sort of physical implementation defeats a primary reason to define TLC algebra in the first place. Instead we define a variation for the join algorithms with the grouping procedure pushed in; we call the resulting operators Nest-Structural-Join and Nest-Value-Join.

Definition 8. **Nest-Structural-Join** $NSJ[LC_l, r, LC_r](S_l, S_r)$: The operator accepts as input two sequences of trees S_l, S_r. As parameters, two LC labels LC_l, LC_r (one for each input sequence) and the desired structural predicate r between them. LC_l must bind to a singleton set of nodes for each tree in S_l and LC_r must bind to the root of each tree in S_r. r indicates the desired structural relationship (parent/child (pc) or ancestor/descendant (ad)). The output is generated using the following algorithm. For each input tree T_l in S_l, and all input trees $T_rm, \ldots, T_rn$ in S_r, where $[r(LC_l, LC_rm) = True), \ldots, (r(LC_l, LC_rn) = True]$, generate one output tree T_o by stitching together T_l and a cluster of $T_rm, \ldots, T_rn$ as specified in $[r(LC_l, LC_rm), \ldots, r(LC_l, LC_rn)]$

The difference from a regular structural join lies in the generation of the output. The regular version creates an output tree for each pair of matching trees from S_l and S_r. Instead, for the nest version each tree in S_l and all matching trees in S_r are returned in one output tree. The example in Figure 14 illustrates the difference between a regular structural join and a nest one.

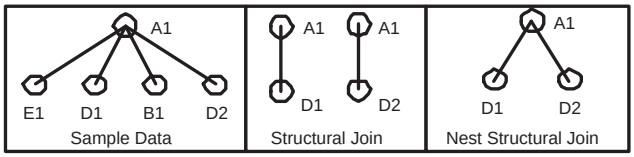

Figure 14: Structural join vs. nest structural join.

The **Nest-Value-Join** $NVJ[LC_l, p, LC_r](S_l, S_r)$ is defined similarly. An output tree consists of one tree in S_l and all matching trees in S_r as described by p. Note that the operator is only left nested. We can also have variations like **Left-Outer-Nest-Structural-Join** and **Left-Outer-Nest-Value-Join**, which output a tree from the left input sequence even if no matching trees from the right sequence are found. We omit the formal definitions due to space restrictions.

Using both regular and Nest-joins as primitives we implement a pattern tree match for an APT using the following simple rules: "-" edges are matched by regular structural and value joins, "?" edges by left-outer-joins, "+" edges are matched using nest-joins and "*" by left-outer-nest-joins. Join order should be considered by an optimizer and describing it is not in the scope of this paper. For our implementation we used a simple bottom-up approach.

6. EXPERIMENTAL EVALUATION

All the experiments were executed using the TIMBER [11] native XML database system. XQuery queries were translated to TLC plans using our algorithm as specified in section 3. The plans utilize our algebraic operators, with no structural rewriting or cost-based optimization performed. We tested the rewrites proposed in section 4 by further extending a few queries and we discuss them in a separate sub-section. For our data set we used the XMark [21, 13] generated documents. Factor 1 produces an XML document that occupies approximately 710MB (479MB for data plus 242MB for indices) when stored in the database. Experiments were executed on an AMD Athlon 1833MHz machine running Windows 2000 professional, with IDE hard drives. The database was set up to use an 128MB buffer pool. All numbers reported are the average of the query execution time over five executions[6].

6.1 The Competition

We wanted to compare TLC against other tree pattern translation approaches that utilize set-at-a-time processing. We chose the popular TAX [9] algebra and the recent and more efficient GTP [4]. Discussing the details of each approach is beyond the scope of this paper. Very briefly we will describe the intuition for each algorithm and our implementation for it.

The TAX algebra plan consists of a sequence of operators that takes a pattern tree as argument. We mapped each pattern tree to a series of structural joins and each TAX operator into corresponding physical operators. Because the TAX plan is a straightforward mapping from logical to physical operators, we will only explain how the logical operators are created for Query Q1. For the FOR/WHERE part TAX will generate a selection with an associated pattern tree in a similar manner to TLC. The selection will be followed by a projection and a duplicate elimination using the same pattern tree, retaining only nodes bound to XQuery variables. The entire subtree is retrieved for such nodes, because it is assumed to be used later in the query. For the RETURN clause TAX will create a selection for every path. Then a join operator will be used to stitch together the RETURN clause paths with the FOR/WHERE parts of the query, joining on the bound variables. TAX does not support annotated edges in its pattern trees, and to compensate for that it uses a grouping procedure to get the semantics for "*" or "+" edges[7]. Note that a simple groupby by a succeeding projection only suffices for the case where TLC would create a single path containing "+" or "*" edges. But if those edges were to span in multiple branches (e.g. multiple arguments in the return, LET, aggregate etc.), the corresponding TAX grouping procedure would involve splitting up the tree, do the grouping and projection for each branch, and then merge the produced paths. Figure 7 shows a TAX-like representation for Q1.

The GTP algorithm uses an alternative approach in creating the execution plan. Instead of creating multiple pattern trees for various subparts of the query (FOR, WHERE, RETURN etc.), an abstract generalized tree is used to capture the semantics for the entire query (contains "?" edge semantics). In the presence of nested queries with joins multiple such trees can be generated. The generalized tree uses special markings for the bound XQuery variables, and annotates the RETURN edges as optional. From that generalized tree the physical plan is generated using structural joins to map each edge. Similar to TAX, aggregates, RETURN paths etc. (everything that corresponds to "+" or "*" pattern tree edge in TLC) are addressed via a grouping procedure that potentially includes splitting the trees, grouping and then merging the results (a DAG-like procedure). But GTP is more efficient than TAX because the generalized tree captures the semantics for the entire query allowing pattern

[6]The highest and the lowest values were removed and then the average was computed

[7]A similar in concept union procedure is used for "?"

tree reuse. Hence many redundant TAX pattern tree matches are avoided. Also there is no need to either bring in memory the subtree for each bound variable node or perform a value join at the end.

We also implemented a navigational algorithm. The algorithm traverses down a path by recursively getting all children of a node and checking them for a condition on content or name before proceeding on the next iteration.

6.2 Tested Queries

We executed dozens of queries : those described in the XMark benchmark as well as our own. In our tests we wanted to check all factors that instigate heterogeneous tree sequences, such as aggregates, return arguments (and number of them), LETs etc. We also wanted to use a diverse set of queries with different selectivity and data materialization costs, enabling us to effectively compare our algorithm against other approaches. We discuss the entire set of XMark queries (x1, . . . , x20), the examples used in the paper (Q1, Q2) and a variation of one XMark query (x10→10a) with a highly selective filter on it.

We used an index on element tag name for all the queries, which returns the node identifiers given a tag name. On all queries that had a condition on content we used a value index, which returns the node ids given a content value. Unfortunately our implementation does not support indices on join values. Results are summarized in Figure 15. The maximum time we allowed a query to run is 10 minutes; if the execution did not finish in that period, we report DNF in the corresponding column.

6.3 Results Discussion

Examining the results presented in Figure 15 we make the following observations. TLC outperforms Navigational(NAV) for every query tested, most times by one or two orders of magnitude. TLC outperforms TAX for every query tested by a large factor and a few times by more than one order of magnitude. TLC outperforms GTP in most cases, a couple of times by up to one order of magnitude.

TLC vs GTP. Since GTP performs better than the other two plans we will start by explaining its differences with TLC. Both TLC and GTP perform the pattern tree match via structural joins, reusing already matched information. For what are TLC "*" and "+" edges, GTP uses a grouping procedure as we described in subsection 6.1. This grouping procedure has the following disadvantages: (i) groupby costs more than nest-joins, (ii) projection has to perform a pattern tree match (via structural joins or navigation) on the grouped results to retrieve the nested nodes, TLC just uses a LC reference, (iii) a DAG-like plan graph has to be used involving splitting and merging each nested path, TLC uses a regular pipelined execution.

For each query path size, value joins and data access costs affect both GTP and TLC the same way. We see a difference in performance when the query uses multiple arguments per return (A/R) clause, aggregates, count, LET bindings and nested clauses (let us call them heterogeneity instigators). So for queries with a single A/R and very few results TLC and GTP perform similarly: such queries are x1, x4, x15 and x16. For queries that have average to lots of produced results and 1 or 2 A/R TLC outperforms GTP up to 2.5 times (for x13): such queries include x2, x3, x13, x14, x17, x18 and x19. When count is used, if the count is small (e.g. counts few elements per tree), like x5, performance difference is average (e.g. 30%), but for multiple counts or big counts TLC performs 2 times faster than GTP. When we mixed this heterogeneity instigators (LET and count and nested queries), the performance difference ranges from 50% to 300%: such queries are x11, x12, x9, Q1 and Q2 (data materialization costs dominate most queries, else we would see even bigger difference). The worst performance is seen for query x10, which involves a nested query on LET using 12 A/Rs in the inner clause. This forces GTP to break the query up into 12 paths, do the grouping procedure and then do another grouping to get the LET binding in the outer part (essentially a nested grouping). The amount of memory needed for such a procedure is big and that forces the computation to run at disk speed (a lot of pages need to be flushed to disk and then read back). The GTP plan for x10, cannot finish in 10 mins, so we used a filtering condition to make it very selective (10a) to show the performance difference of about one order of magnitude vs TLC.

	TLC	GTP	TAX	NAV	Comments
x1	0.05	0.06	0.17	29.03	1 A/R, single OT
x2	0.61	0.96	12.54	25.88	1 A/R, lots OT
x3	2.16	2.95	6.45	26.94	J, 2 A/R, avg OT
x4	0.30	0.33	1.39	34.91	1 A/R, two OT
x5	0.17	0.21	1.06	29.78	small count, 1 A/R
x6	0.11	0.22	0.90	128.3	big count, '//'
x7	0.49	0.98	1.91	131.1	3 big counts, '//'
x8	4.88	6.72	28.56	43.52	J, LET, 2 A/R
x9	7.41	11.80	83.28	82.63	2J, LETs, 2 A/R
x10	74.3	DNF	DNF	89.24	LET, 12 A/R, lots OT
x11	9.95	13.56	45.67	52.70	count, LET, lots OT
x12	4.50	6.79	15.67	49.24	count, LET, avg OT
x13	0.23	0.62	1.89	30.88	2 A/R, avg OT
x14	2.26	2.78	6.44	134.4	'//', contains on desc
x15	0.97	0.97	2.70	87.05	long path, return $var
x16	1.16	1.27	2.89	87.95	long path, 1 A/R
x17	0.36	0.59	1.65	27.89	1 A/R, lots OT
x18	0.11	0.27	1.22	24.13	1 A/R, lots OT
x19	4.18	5.08	12.39	129.4	//, 2 A/R, sort, lots OT
x20	0.66	0.78	5.88	27.47	4 counts
Q1	2.35	5.14	19.21	132.5	'//', J, count, 2 A/R
Q2	2.70	8.88	46.34	137.4	//, J,count,2 A/R, LET
10a	3.03	33.69	79.92	87.46	LET, 12 A/R, few OT

Figure 15: Execution time in seconds for XMark data size factor 1. Algorithms used: Navigational(NAV), TAX, GTP and TLC. We used XMark XQueries (x1, . . . , x20), queries Q1, Q2 from the paper examples, and 10a as a variation of x10 with a selective filter applied on it. The abbreviations used in the comments are: A/R = Arguments per Return clause, OT = Output Trees, J = Value Join.

TLC vs TAX. TAX, when compared with TLC, has all the drawbacks that GTP has, plus a few extra of its own. TAX does not reuse pattern tree matches, it materializes results early (brings in subtrees for bound variables) and needs to perform a join to process the RETURN clause (aside from the grouping procedure). This makes the behavior of TAX much worse than that of GTP. The best case for TAX is when either the queries produce few results with 1 A/R and no other heterogeneity instigator or the queries have a simple selection and dominating data access cost. Such cases are queries x1, x3, x14, x15, x16 and x19, where TLC is 2 to 3 times faster than TAX. If the query includes multiple heterogeneity instigators, TAX suffers a lot. The early materialization imposes a penalty for carrying data nodes through all the groupings for counts, join, LETs etc. Also due to this reason, nested queries perform poorly for TAX: notice how TLC wins by over an order of magnitude for x9, x10 and Q2.

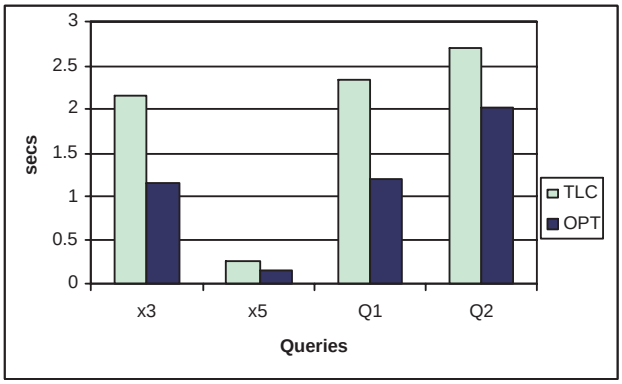

Figure 16: Execution time in seconds. Presenting the regular TLC plan vs. the optimized (OPT) plan, produced by performing the Flatten and Shadow rewrites.

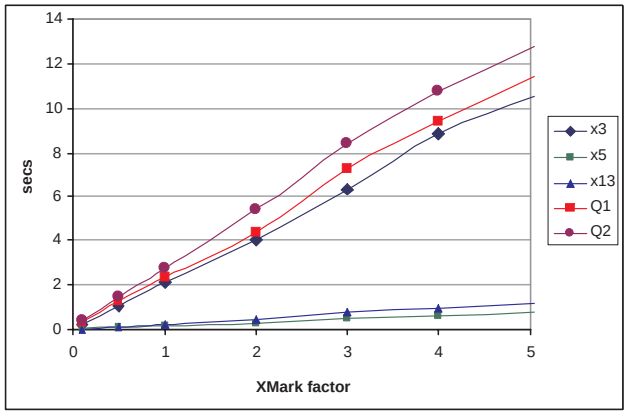

Figure 17: Execution time in seconds for XMark size factor 0.1 through 5. Queries x3, x5, x13, Q1, Q2 are plotted.

TLC vs Navigational. Navigational (NAV) behaves very differently than TAX and GTP. In general, data materialization cost does not affect NAV much since it has already paid the cost of getting all children[8]. So having to deal with nested queries, LETs and multiple A/R is not a problem. Instead we found that NAV highly depends on the path size and on the number of children for each data node. The smaller the path and the number of children the better the algorithm behaves. For example, x3 has a path *site/open_auctions/open_auction* which corresponds to 1/6/many elements. All of the *open_auction* elements *have* to be considered for this query (all of many) by *all* algorithms. So this is one of the better cases for a navigational plan. NAV suffers when the query uses '//' like x6, x14, x19, Q1 and Q2. Navigational is affected by selectivity a lot, it has to do the same amount of iterations even if the query were to produce zero results, hence highly selective queries pose a problem for navigation. So for x10 where nothing is filtered out, TLC performs very close to NAV; the data materialization cost dominates the query. But for a query like x1, which is highly selective, TLC performs 2 orders of magnitude better. The worst case scenario for navigation is the usage of counts. TLC performs the count without touching the data in a fraction of a second whereas NAV has to iterate over all nodes: such examples are query x5 where TLC is 2 orders of magnitude better and x6 that uses '//' besides counts and TLC is 3 orders of magnitude better.

6.4 Flatten and Shadow/Illuminate

The proposed Flatten and Shadow/Illuminate rewrites as described in section 4 are not applicable to all queries. We present four queries, x3, x5, Q1 and Q2, where the rewrites were applicable. The results are shown in Figure 16. The figure shows the performance of each query for the regular TLC plan and the optimized (OPT) plan. As we can see, the optimizations can lead to a big performance increase: after performing the rewrites TLC plans can be up to 2 times faster. The performance difference is due to the eliminated redundant structural joins and data accesses. The higher the number of eliminated joins the better the performance will be.

6.5 Scalability

We tested all our queries to see how TLC produced plans scale using our implementation. We used a variety of XMark data size factors from 0.1 (approx. 67MB combined data plus indexes space) up to factor 5 (approx. 3.5GB combined data plus indexes space). Plotting 23 queries would only make the figure impossible to read. So we used queries x3, x5, Q1, Q2 (already talked about in the previous section) and x13. The results are seen in Figure 17; the produced TLC plans scale linearly with size. Queries x5 and x13 are simple selections where the dominating cost is that of the structural joins – a value index was used in x5 to satisfy the predicate. Queries x3, Q1 and Q2 have value joins and Q2 is also a nested query. Since our implementation did not support value join indexes, data access on the join value is the dominating cost. Please note that value join queries scale linearly because our node identifiers allow us to use the *sort-merge-sort* algorithm mentioned in Section 5.1.

[8]In our storage nodes are clustered with their children. So the disk cost of getting all children ids is almost the same as getting all children ids and their values.

7. RELATED WORK

There are three major implementation approaches to XML data management and XQuery evaluation. Mapping to relational [7, 17, 6, 22, 15], native navigational-based [16], and native algebraic-based [11, 12, 14, 20]. Most previous work on algebraic native XQuery implementation has focused on efficient evaluation of XPath expressions via structural joins [1], holistic joins [3] and optimal ordering of structural joins for pattern matching [19, 5]. The Niagara [12] system makes extensive use of structural joins for pattern matches, as does TIMBER [11]. We feel that these systems could benefit from the nest-joins variations we introduced in our implementation as an important primitive.

There is no universally accepted XML algebra. In the past there have been proposals in the literature, divided into two major approaches: node algebras [10, 18] and tree algebras [9, 4]. The former manipulate sets of XML elements (nodes), but have a problem with the inclusion structure of XML. They either lose it or are forced to maintain the subtree under each node (Niagara). The latter class addresses the problem by using the more efficient concept of a pattern-tree match. Witness trees produced after the match have uniform structure (identical to the pattern tree) and individual elements in the match result can be named and accessed in the algebraic operators. But the semantics of XQuery demand sub-trees of the original XML tree to be restructured and heterogeneous tree sequences to be generated for the query to be evaluated correctly (e.g. aggregates, LET bindings, RETURN bindings etc.). Tree algebras like TAX [9] and GTP [4] rely on a grouping procedure to restructure the repeated elements under some node. Then to identify the nodes of interest, such algebras have to navigate the in-memory structure or perform another pattern tree match. Although this generates the correct solution, it performs poorly when

the query workload contains several such cases[9]. TLC solves this problem by using Annotated Pattern Trees (APTs) to construct the nested elements and Logical Class (LC) bindings to sets of nodes to identify the nodes of interest and allow the operators to use them. We feel that any tree algebra can be extended appropriately to use APTs and LCs and have the same benefits with TLC when dealing with heterogeneous tree sequences.

Recently the "Dynamic-Intervals" paper [6] tries to mix the relational mapping implementation with the XML-specific algebraic approach of native XML databases. They make use of the well-known interval encoding of trees and then assign these intervals dynamically for intermediate results. XQueries are translated to an extend SQL that includes structural-join like primitives. A direct experimental comparison with their approach is difficult since their algorithm is tied with their implementation and focuses on the dynamic interval assignment. Doing an informal system comparison (both implementation and algorithm) as described in their paper, we outperform them by up to an order of magnitude (see queries x8, x9, x13 that use no value or join index as they did). Given the different platforms used, such comparison is definitely ad hoc. However, it serves to give a general indication. We feel that even their system can benefit by applying TLC on top of their implementation as long as they support primitives like structural-joins and nest-structural-joins and add support for logical classes.

8. CONCLUSION

The flexibility of XML poses a significant challenge to query processing: it is hard to perform set-oriented bulk operations on heterogeneous sets. In this paper we have proposed the TLC algebra as an effective means to address this problem through a novel notion of logical classes and related logical class reduction. Through this, not only do we enable algebraic operators to work with heterogeneous sets, but we also eliminate redundant computation required by previous approaches. In addition to proposing the TLC algebra, our contributions include: the presentation of an algorithm to translate a significant fragment of XQuery into TLC algebra, the development of an efficient implementation of TLC algebra and a careful experimental evaluation that shows the substantially superior performance of our TLC implementation compared to other implementations of the same queries via different algebras.

9. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE Conf.*, Mar. 2002.

[2] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. Working Draft. `http://www.w3.org/TR/xquery`.

[3] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD Conf.*, 2002.

[4] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *Proc. VLDB Conf.*, Sep. 2003.

[5] C.Zhang, J.Naughton, D.Dewitt, Q.Luo, and G.Lohman. On supporting containment queries in relational database management systems. In *Proc. SIGMOD Conf.*, 2001.

[6] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proc. SIGMOD Conf.*, Jun. 2003.

[7] D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3), 1999.

[8] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C.Yu. TIMBER: A native XML database. *VLDB Journal*, 11(4), 2002.

[9] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. DBPL Conf.*, Sep. 2001.

[10] B. Ludascher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven evaluation of virtual mediated views. In *Proc. EDBT Conf.*, Mar. 2000.

[11] U. of Michigan. The TIMBER project. `http://www.eecs.umich.edu/db/timber`.

[12] U. of Wisconsin. The Niagara internet query system. `http://www.cs.wisc.edu/niagara/`.

[13] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. VLDB Conf.*, 2002.

[14] H. Schoning. Tamino - A DBMS designed for XML. In *Proc. ICDE Conf.*, 2001.

[15] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. VLDB Conf.*, 1999.

[16] J. Simeon and M. F. Fernandez. Galax, an open implementation of XQuery. `http://db.bell-labs.com/galax/`.

[17] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. SIGMOD Conf.*, 2002.

[18] S. D. Viglas, L. Galanis, D. J. DeWitt, D. Maier, and J. F. Naughtonn. Putting XML query algebras into context. `http://www.cs.wisc.edu/niagara/`.

[19] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proc. ICDE Conf.*, Mar. 2003.

[20] X-Hive Corp. X-Hive/DB native XML storage. `http://www.x-hive.com/`.

[21] XMark, an XML benchmark project. `http://www.xml-benchmark.org/`.

[22] X. Zhang, B. Pielech, and E. A. Rundensteier. Honey, i shrunk the XQuery! — an XML algebra optimization approach. In *Workshop on Web Information and Data Management*, 2002.

[9] such a query workload is becoming all too common