From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery

Zhimin Chen[1] **H.V. Jagadish**[2] **Laks V. S. Lakshmanan**[1]
Stelios Paparizos[2]

[1] Univ. of British Columbia, {zmchen, laks}@cs.ubc.ca
[2] Univ. of Michigan, {jag, spapariz}@eecs.umich.edu

Abstract

XQuery is the de facto standard XML query language, and it is important to have efficient query evaluation techniques available for it. A core operation in the evaluation of XQuery is the finding of matches for specified *tree patterns*, and there has been much work towards algorithms for finding such matches efficiently. Multiple XPath expressions can be evaluated by computing one or more tree pattern matches.

However, relatively little has been done on efficient evaluation of XQuery queries as a whole. In this paper, we argue that there is much more to XQuery evaluation than a tree pattern match. We propose a structure called *generalized tree pattern* (GTP) for concise representation of a whole XQuery expression. Evaluating the query reduces to finding matches for its GTP. Using this idea we develop efficient evaluation plans for XQuery expressions, possibly involving join, quantifiers, grouping, aggregation, and nesting.

XML data often conforms to a schema. We show that using relevant constraints from the schema, one can optimize queries significantly, and give algorithms for automatically inferring GTP simplifications given a schema. Finally, we show, through a detailed set of experiments using the TIMBER XML database system, that plans via GTPs (with or without schema knowledge) significantly outperform plans based on navigation and straightforward plans obtained directly from the query.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

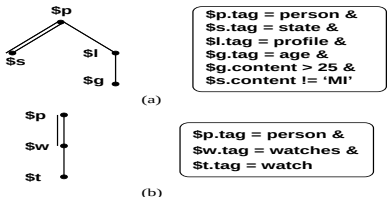


Figure 1: Two Example Tree Pattern Queries: (a) P_1 and (b) P_4. Single (double) edge represents parent-child (ancestor-descendant) relationship.

1 Introduction

XQuery is the current de facto standard XML query language. Several XQuery implementation efforts have been reported around the world. A key construct in most XML query models is the so-called *tree pattern (query)* (TP(Q)), which is a tree T with nodes labeled by variables, together with a boolean formula F specifying constraints on the nodes and their properties, including their tags, attributes, and contents. The tree consists of two kinds of edges – parent-child (pc) and ancestor-descendant (ad) edges. Fig. 1(a)-(b) shows example TPQs; in (b), we call node $w an ad-child of $p, and a pc-parent of $t.

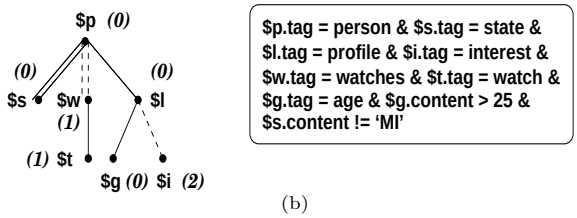
The semantics of a TPQ $P = (T, F)$ is captured by the notion of a *pattern match* – a mapping from the pattern nodes to nodes in an XML database such that the formula associated with the pattern as well as the structural relationships among pattern nodes is satisfied. The TPQ in Fig. 1(a) (against the auction.xml document of the XMark benchmark [24]) matches person nodes that have a state subelement with value $\neq$ 'MI' and a profile with age > 25. The state node may be any descendant of the person node.

Viewed as a query, the answer to a TPQ is the set of all node bindings corresponding to valid matches. The central importance of TPQs to XML query evaluation is evident from the flurry of recent research on efficient evaluation of TPQs [26, 2, 7].

While XQuery expression evaluation includes the matching of tree patterns, and hence can include TPQ evaluation as a component, there is much more to

```
FOR $p IN document("auction.xml")//person, $l IN $p/profile
WHERE $l/age > 25 AND $p/state != 'MI'
RETURN <result>{$p//watches/watch}{$l/interest} </result>
```

(a)



(b)

Figure 2: An Example XQuery query and corresponding Generalized Tree Pattern Query. Solid (dotted) edges = compulsory (optional) relationship. Group numbers of nodes in parentheses.

XQuery than simply TPQ. In particular, the possibility of quantification in conditions (e.g., EVERY), the possibility of optional elements in a return clause, and the many different forms of return results that can be constructed using just slightly differing XQuery expressions, all involve much more than merely obtaining variable bindings from TPQ evaluation. To facilitate an efficient evaluation of XQuery queries, we propose the notion of a *generalized tree pattern* (GTP). Intuitively, a GTP provides an abstraction of the work that needs to be done toward query evaluation, and provides clues for doing this work while making as few passes over the input data as possible. As a preview, Fig. 2(b) shows a sample query (against the auction.xml document of the XMark benchmark [24]) as well as the associated (rather simple in this case) GTP. The GTP has solid and dotted edges. Solid edges represent mandatory relationships (pc or ad) just like edges of a TPQ. Dotted edges denote optional relationships: e.g., `$i` optionally may be a child of `$l`, and `$w` optionally may be a descendant of `$p`. The GTP can be informally understood as follows: (1) Find matches for all nodes connected to the root by only solid edges. (2) Next, find matches to the remaining nodes (whose path to the GTP root involves one or more dotted edges), *if they exist*.

We will show later that GTPs can be used to answer queries involving quantifiers, grouping, aggregation, and nesting. More complex examples of a GTP appear in Fig. 7 and 8.

We make the following contributions in this paper.

- We propose the notion of a generalized tree pattern (Section 3).
- We show GTPs contain enough information for evaluating XQuery expressions. Specifically, we give an algorithm for translating an XQuery expression conforming to the grammar in Fig. 9 into a GTP (Section 3.4).
- We present a physical algebra for query evaluation, appropriate for a native implementation of XQuery. We give an algorithm that translates a GTP into an equivalent expression in our physical algebra (Sections 4.1, 4.1.1).

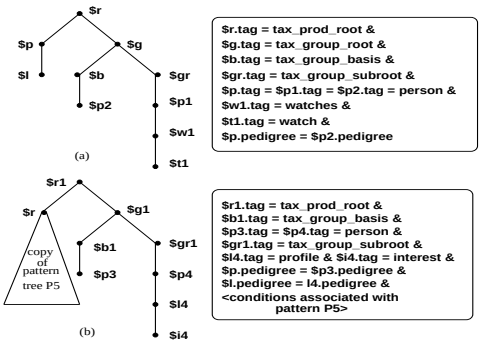


Figure 3: Two TPQs: (a) P_6; (b) P_8.

- When schema information is available, we show how it can be exploited to remove redundant parts of the GTP, and to eliminate unnecessary operators in the physical query plan (Section 5).

We consider alternative evaluation strategies and show with a thorough experimental study that plans generated by using GTPs and then translating them into physical algebraic expressions far outperform alternative evaluation strategies in most cases (Section 6). We also demonstrate the savings obtained by incorporation of schema knowledge in query optimization. In Section 7, we discuss related work, and conclude the paper in Section 8.

2 Motivation

In this section, we motivate GTPs by illustrating that straightforward approaches to evaluating queries can lead to very inefficient plans.

Example 1 (Straightforward is Inefficient) Consider again the query given in Fig. 2(a). This is one of the simplest queries involving minimal features from XQuery, and indeed the simplest query discussed in this paper. This query cannot be computed using a *single* TPQ: the most obvious candidate is the TPQ obtained from Fig. 2(b) by turning dotted edges into solid ones. However, such a tree will produce, in each match result, a person element and precisely one watch subelement (and a profile element with precisely one interest subelement). A person with no watch subelements will not even appear in the output (because the match fails on the watch subelement). A person with two watches appears twice in two separate results, once with each watch sub-element. (What XQuery would produce instead is the person appearing once, with *both* watch subelements, in the result).

Consider instead a TPQ with only the solid edges in Fig. 2(b). The match results will produce exactly the correct set of bindings for the nodes in the pattern (person, profile, etc.). However, we now have additional work to do to incoroporate the watch, interest, etc. optional sub-elements. For each match result, we could navigationally find these sub-elements.

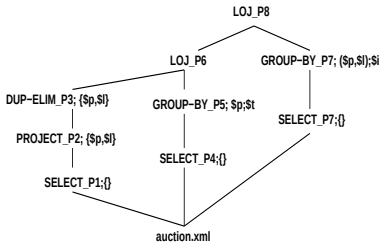


Figure 4: TAX operator tree for the query of Fig. 2(a). LOJ = left outer-join.

However, this can become quite expensive, particularly since return clauses in XQuery expressions can often express quite complex relationships to the bound variables. Previous research [26, 2] has shown that set-oriented structural join (tree pattern match) computations are most often substantially more efficient than navigational approaches. Our own experiments (reported in Section 6) also corroborate this.

Yet, a third possibility is to find the correct set of bindings using the solid edge TPQ as in the preceding paragraph, but then use set-oriented manipulation to populate the remaining optional (and possibly repeating) nodes in the pattern. Doing so requires a sequence of multiple TPQ matches, and grouping of partial answers to construct each result tree with multiple watch elements under person and multiple interest elements under profile. A schematic description of this procedure is shown in Fig. 4, using operators in the TAX algebra [12].[1] While the details of this algebra are orthogonal to this paper, it is sufficient to note: (i) the operators have a flavor similar to relational algebra but they make use of TPQs and pattern match to access nodes of interest in trees, and (ii) the physical plan corresponding to Fig. 4 is quite complex and inefficient. E.g., similar tree patterns are repeatedly matched. [14] gives a full explanation of all the steps required. To sum, a correct set-oriented evaluation of XQuery is possible, but can get quite complicated even for simple XQueries. (We will quantify the performance cost of this complication in Section 6.)

The GTP of Fig. 2(b), for this simple example, is interpreted to produce as match results precisely the set of answers XQuery semantics would expect.

3 Generalized Tree Patterns

In this section, we introduce generalized tree patterns (GTP), define their semantics in terms of pattern match, and show how to represent XQuery expressions as GTPs. For expository reasons, we first define the most basic type of GTP and then extend its features as we consider more complex fragments of XQuery.

Definition 1 [Basic GTPs] A *basic generalized tree pattern* is a pair $G = (T, F)$ where T is a tree and

[1]The TPQs used in Fig. 4, including P_6 and P_8 from Fig. 3, are explained in Section 6.1, BASE.

$\wedge$	$\bot$	1	0
$\bot$	$\bot$	1	0
1	1	1	0
0	0	0	0

$\vee$	$\bot$	1	0
$\bot$	$\bot$	1	0
1	1	1	1
0	0	1	0

$\neg$	$\bot$	1	0
	$\bot$	0	1

Figure 5: Extension to handle '*undefined*' truth value.

F is a boolean formula such that: (i) each node of T is labeled by a distinct variable and has an associated group number; (ii) each edge of T has a pair of associated labels $\langle x, m\rangle$, where $x \in \{pc, ad\}$ specifies the axis (parent-child and ancestor-descendant, respectively) and $m \in \{mandatory, optional\}$ specifies the edge status; and (iii) F is a boolean combination of predicates applicable to nodes. [2] ∎

Fig. 2(b) is an example of a (basic) GTP. Rather than edge labels, we use solid (dotted) edges for mandatory (optional) relationship and single (double) edges for pc (ad) relationship.

We call each maximal set of nodes in a GTP connected to each other by paths not involving dotted edges a *group*. Groups are disjoint, so that each node in a GTP is member of exactly one group. We arbitrarily number groups, but use the convention that the the group containing the for clause variables (including the GTP root) is group 0. In Fig. 2(b) group numbers are shown in parantheses next to each node.

Let $G = (T, F)$ be a GTP and $\mathcal{C}$ a collection of trees. A *pattern match* of G into $\mathcal{C}$ is a partial mapping $h : G \rightarrow \mathcal{C}$ such that:

- h is defined on all group 0 nodes.
- if h is defined on a node in a group, then it is necessarily defined on all nodes in that group.
- h preserves the structural relationships in G, i.e., whenever h is defined on nodes u, v and there is a pc (ad) edge (u, v) in G, then $h(v)$ is a child (descendant) of $h(u)$.
- h satisfies the boolean formula F.

Observe that h is partial matching: elements connected by optional edges may not be mapped. Yet, we may want the mapping as a whole to be valid in the sense of satisfying the formula F. To this end, we extend boolean connectives to handle the '*undefined*' truth value, denoted $\bot$.[3] Fig. 5 shows the required extension. In a nutshell, the extension treats $\bot$ as an identity for both $\wedge$ and $\vee$ and as its own complement for $\neg$.

In determining whether a pattern match satisfies the formula F, we set each condition depending on a node not mapped by h to $\bot$ and use the extensions to connectives in Fig. 5 to evaluate F. Iff it evaluates to *true*, we say h satisfies F. The optional status of edges is accounted for by allowing groups (other than 0) to be not mapped at all, while still satisfying F. As an

[2]Additionally, each node corresponding to a FOR variable, also has a number indicating its order in the FOR clause, a detail suppressed for brevity.

[3]It turns out standard 3-valued logics like that of Kleene do not work for our purposes.

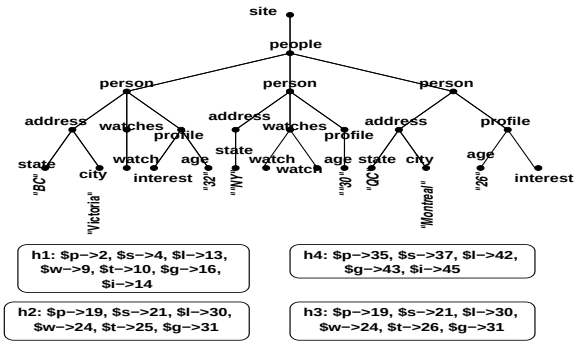


Figure 6: (a) Sample XML data. (b) Pattern matches of GTP of Fig. 2(b).

example, consider a pattern match h that maps only nodes \$p, \$s, \$l, \$g, \$i in Fig. 2(b) and satisfies only conditions depending on these nodes. Setting all other conditions to $\perp$, it is easy to check h does indeed satisfy the formula in Fig. 2(b). We call a pattern match of a GTP *valid* if it satisfies the boolean formula associated with the GTP.

Fig. 6 shows a sample XML document (in tree form) and the set of valid pattern matches of the GTP of Fig. 2(b) against it. Note that h_2, h_3 are not defined on group 2, while h_4 is not defined on group 1. Also, matches h_2 and h_3 belong to the same logical group since they are identical except on pattern node \$$t$.

3.1 Join Queries

A join query clearly warrants one GTP per document mentioned in the query. However, we need to evaluate these GTPs in sync, in the sense that there are parts in different GTPs that must both be mapped or not at all. Fig. 8 shows a (nested) query involving join and a corresponding GTP. It is discussed at length in Section 3.3. The appendix gives another example.

3.2 Grouping, Aggregation, and Quantifiers

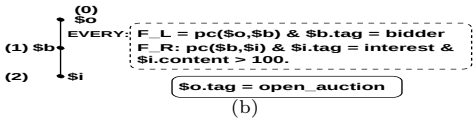
Conventional value aggregation in itself does not raise any special issues for GTP construction. Structural aggregation, whereby collections are grouped together to form new groups, is naturally handled via nested queries, discussed in Section 3.3. So we next focus just on quantifiers.

Basic GTPs can already handle SOME quantifier, since an XQuery expression with SOME can be rewritten as an one without it. Handling EVERY quantifier requires an extension to GTPs.

Definition 2 [Universal GTPs] A *universal* GTP is a GTP $G = (T, F)$ such that some solid edges may be labeled 'EVERY'. We require that: (i) a node with an incident EVERY edge is reachable from the GTP root by a path containing only solid edges (some of which may be EVERY edges), (ii) the GTP includes a pair of formulas associated with an EVERY edge, say F_L and F_R, that are boolean combinations of predicates applicable to nodes, including structural ones,

```
FOR $o IN document("auction.xml")//open_auction
WHERE EVERY $b in $o/bidder SATISFIES $b/increase > 100
RETURN <result> {$o} </result>
```

(a)



(b)

Figure 7: An example universal query and corresponding universal GTP.

and (iii) nodes mentioned in F_L should be in a separate group by themselves. ∎

Example 2 (Universal GTP) Fig. 7 shows a query with universal quantifier and a corresponding universal GTP. The GTP codifies the condition that for every bidder \$b that is a subelement of the open_auction element \$o, there is an increase subelement of the bidder with value > 100. ∎

The formula associated with the EVERY edge represents the constraint $\forall \$b : [F_L \rightarrow \exists \$i : (F_R)]$, for the above example, $\forall \$b : [\$b.tag = \text{bidder} \ \& \ pc(\$o, \$b) \rightarrow \exists \$i : (\$i.tag = \text{interest} \ \& \ pc(\$b, \$i) \ \& \ \$i.\text{content} > 100)]$.

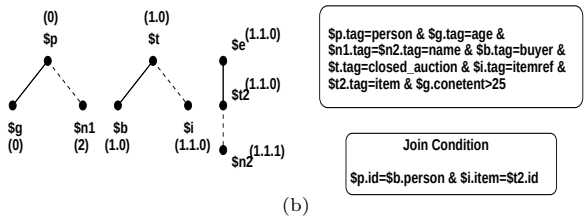
3.3 Nested Queries

We use a simple device of a hierarchical group numbering scheme to capture the dependence between a block and the corresponding outer block in the query.

Example 3 (Nested Query) Consider the nested query in Fig. 8(a). Corresponding to the outer for/where clause, we create a tree with root \$p (person) and one solid pc-child \$g (age). They are both in group 0. We process the inner FLWR statement binding \$a. Accordingly, we generate a tree with root \$t (closed_auction) with a solid pc-child \$b (buyer). Put these nodes in group 1.0, indicating they are in the next group after group 0, but correspond to the for/where part of the nested query. Finally, we process the return statement and the nested query there. For the for/where part, we create a tree with root \$e (europe) with a solid pc-child \$t2 (item), both being in group 1.1.0. We also create a dotted pc-child \$i (itemref) for \$t, corresponding to the join condition \$t/itemref/@item=\$t2/@id in the corresponding where clause. Since it's part of the for clause above, we assign this node the *same* group number 1.1.0. The only return argument of this innermost query is \$t2/name, suggesting a dotted pc-child \$n2 (name) for node \$t2, which we add and put in group 1.1.1. We also create a dotted pc-child \$i (itemref) for \$t, corresponding to the join condition \$t/itemref/@item=\$t2/@id in the inner where. Finally, exiting to the outer return statement, we see the expressions \$p/name/text() and \$a. The first of these suggests a dotted pc-child \$n (name) for \$p, which we add and put in group 2. The second of these, \$a, corresponds to the sequence of european item names bound

```
FOR $p IN document("auction.xml")//person
LET $a :=
    FOR $t IN document("auction.xml")//closed_auction
    WHERE $p/@id=$t/buyer/@person
    RETURN <item>
           {FOR $t2 IN document("auction.xml")//europe/item
           WHERE $t/itemref/@item=$t2/@id
           RETURN {$t2/name}}
           </item>
WHERE $p//age>25
RETURN <person name=$p/name/text()> $a </person>
```

(a)



(b)

Figure 8: An Example query with nesting & join and corresponding GTP.

to it by the LET statement, and as such is covered by the node $n2. The GTP we just constructed is shown in Fig. 8(b). ∎

In general, we can only match a group (e.g., 1.1.0) after its "parent" group (1.0) is matched. As usual, either all nodes in a given group must be matched or none at all. For this example, the sequence in which matches should be determined for different groups is concisely captured by the expression 0[2][1.0[1.1.0[1.1.1]]], where [G] means the groups mentioned in G are matched optionally.

3.4 Translating XQuery to GTP

Putting the above ideas together, we obtain Algorithm GTP for translating an XQuery query into a corresponding GTP. While most of function-free XQuery can be handled by this algorithm, we restrict our exposition here to the simplified, yet substantially expressive, fragment of XQuery, captured by the grammar in Fig. 9.

```
FLWR ::= ForClause LetClause WhereClause ReturnClause.
ForClause ::= FOR $fv_1 IN E_1, ..., $fv_n IN E_n.
LetClause ::= LET $lv_1 := E_1, ..., $lv_n := E_n.
WhereClause ::= WHERE φ(E_1,...,E_n).
ReturnClause ::= RETURN {E_1}...{E_n}.
E_i ::= FLWR | XPATH.
```

Figure 9: Grammar for XQuery Fragment.

The algorithm has a global parsing environment *ENV* for bookkeeping the information collected from parsing, including, e.g., variable name-pattern node association, GTP-XML document source association, etc. It also uses a helper function *buildTPQ(xp)*, where xp is an (extended)[4] XPath expression, that builds a part of GTP from the xp. If xp starts with the built-in `document` function, a new GTP is added to *ENV*; if xp starts with a variable, the pattern node associated with that variable is looked up and the new part resulting from xp starts from it. The function examines

[4]XQuery allows XPath expressions extended with variables.

```
Algorithm GTP
Input:  a FLWR expression Exp, a context group number g
Output:  a GTP or GTPs with a join formula
if (g's last level != 0)
   let g = g + ".0";
foreach ("For $fv in E") do
   parse(E,g);
let ng = g;
foreach ("Let $lv := E") do{
   let ng = ng + 1;
   parse(E,ng);
}
/* processing WhereClause */
foreach predicate p in φ do {
   if (p is "every E_L satisfies E_R"){
      let ng = ng + 1;
      parse(E_L,ng);
      let F_L be the formula associated
         with the pattern resulted from E_L;
      let ng = ng + 1;
      parse(E_R,ng);
      let F_R be the formula associated with
         the pattern resulted from E_R;
   } else {
      foreach E_i as p's argument do
         parse(E_i,g);
      add p to GTP's formula or the join formula;
      if (p is "count($n')>c" && c >= 0){
         g'=group($n');
         if (g is the prefix of g')
            set the group number of all nodes in g' to g;
      }
      if (p refers to max(/min/avg/sum)($n') and $n
            && group($n)==g){
         g'=group($n');
         if (g is the prefix of g')
            set the group number of all nodes in g' to g;
      } } }
/* processing ReturnClause */
foreach "{E_i}" do {
   let ng = ng + 1;
   parse(E,ng);
}
end Algorithm
procedure parse
Input:  FLWR expression or XPath expression E,
   context group number g
Output:  Part of GTP resulting from E
if (E is FLWR expression)
   GTP(E,g);
else buildTPQ(E);
end procedure
```

Figure 10: Algorithm `GTP`

each location step in xp, creates a new edge and a new node, annotates the edge as pc(or ad, cp, da) as appropriate, according to the axis of the location step and adds a predicate about the node's tag and/or its properties. It returns the distinguished node of xp. Any filter expressions in xp are handled in a way similar to the where clause is, except they are simpler.

Group numbers produced for GTP nodes are strings of numbers. The algorithm accepts a group number as its parameter, which is initialized to the empty string when invoking the algorithm for the first time. We use the shorthand $g + ".x"$ for appending the number "x" to the string g, and $g+1$ for adding 1 to the rightmost number in the string g.

4 Translating GTP Into an Evaluation Plan

The main motivation behind GTP is that it provides a basis for efficient implementation. This is achieved

by: (i) avoiding repeated matching of similar tree patterns and (ii) postponing the materialization of nodes as much as possible. We first discuss a physical algebra for XML, each of whose operators is likely to be available as an access method in any XML database.

4.1 Physical Algebra

Every physical algebra operator maps one or more sequences of trees to one sequence of trees. Except where there is an explicit sorting order specified for the output, we retain in the output sequence the order of the input sequence, captured by means of order of node id's.

Index Scan:($IS_p(S)$) : For each input tree in S, output each node satisfying the specified predicate p using an index.

Filter: $F_p(S)$: Given a sequence of trees S, output only the trees satisfying the filter predicate p. Order is preserved.

Sort: $S_b(S)$: Sort the input sequence of trees S based on the sorting basis b. The output order sequence reflects the sorting procedure (e.g., by value or by node id order of specified node).

Value Join: $J_p(S_1, S_2)$: Perform a value-based comparison on the two input sequences of trees via the join predicate p, using nested loops or sort-merge join. The output sequence order is based on the left S_1 input sequence order. Variants include left-outer join with its standard meaning.

Structural Join: $SJ_r(S_1, S_2)$: The input tree sequences S_1, S_2 must be sorted based on the node id of the desired structural relationship. The operator joins S_1 and S_2 based on the structural relationship r between them (ad or pc) for each pair. The output is sorted by S_1 or S_2 as needed. . Variations include: the Outer Structural Join (OSJ) where all of S_1 is included in the output, Semi Structural Join (SSJ) where only S_1 is retained in the output, Structural Anti-Join (ASJ) where the two inputs are joined based on one *not* being the ad/pc-relative of the other, and combinations.

Group By: $G_b(S)$: Assumes the input is sorted on the grouping basis b. Group trees based on the grouping basis b. Create output trees containing dummy nodes for grouping root, sub-root and basis and the corresponding grouped trees. Order is retained.

Merge: $M(S_1, \ldots, S_n)$: The S_j's are assumed to have the same cardinality, say k. Perform a "naïve" n-way merge of the input tree sequences. For each $1 \leq i \leq k$, merge tree i from each input under an artificial root and produce an output tree. Order is preserved.

While the majority of the physical algebra operators are what one would expect, (including structural joins, which are known to be important for XML query processing), the Merge operator is worth a special mention. It is very simple in terms of what it does, but critical to our ability to stitch together multiple groups of optional return elements.

4.1.1 Translating GTP to Physical Plans

The Evaluation algorithm translates GTP into a physical plan. The plan is a DAG, in which each node is a physical operator or is an input document. To match EVERY edges in the GTP to structural anti-join, it converts them into "forbidden" edges using the transformation $\forall \$x : [F_L \rightarrow \exists \$y : F_R] \equiv \neg\exists \$x : [F_L \& \neg\exists \$y : F_R]$. It also ignores the issue of value join order, assuming it can borrow such techniques from the relational domain.

The algorithm uses a helper function $findOrder(SJs, \$n)$, where SJs is a list of structural joins, $\$n$ is a pattern node and may be optional. The function rearranges the order of SJs such that executing SJs in the order of SJs_1, ..., SJs_n is the optimal order. After executing the SJs, if $\$n$ is present, the returned witness trees are in the ascending order of $\$n$'s node id. The helper function $getGroupBasis(g)$ takes a group number g as its parameter and returns an appropriate nested sequence of pattern nodes that are related to the for variables in all the 0 groups that are prefix to g. (Abusing terminology, we say group g is a prefix to group g' provided g ends with 0 and after excluding the 0 at its end, it is a prefix to g'.) For instance, assume that $\$n_{01}$ and $\$n_{02}$ are the nodes related to fv_1 and fv_2 in group 0, respectively, and $\$n_{11}$ is the node related to the only fv in group 1.0, then getGroupBasis(1.1) returns $< \$n_{01}, \$n_{02}, < \$n_{11} >>$. The helper function $getGroupEvalOrder(G)$ returns the evaluation order of the groups in a GTP G. Basically, the order it returns is the alphabetical order of the group number, except that in the presence of a forbidden edge, the group under the forbidden edge is evaluated before the group above the forbidden edge. The algorithm also prepares the input stream for a pattern node $\$n$ from the XML document using a tag index scan operator, or a value index scan operator if there is such value index and there is a predicate $p(\$n, c)$ in the formula. In such case, there may be a sorting operator following the value index scan. For instance, if a node has a constraint $\$n.tag = age \& \$n.content > 40$, the plan to fetch the data is $Filter_{content>25}(IS_{tag=age}(TagIndex))$ if there is no value index on age, or $Sort(IS_{content>25}(ValueIndexonage))$ otherwise.

Each intermediate result of an operator in the plan has a record about what pattern nodes are bound in the output after executing the operator, whether the output of operator is duplicate free, and whether, if any, the output maintains a sorting order on some nodes' node id's. The output of some operators, e.g., SJ (structural join) or S (sort), maintain some node id order, while some, e.g., VJ (value-based join), do not. The algorithm keeps track of all the output structure record of every operator when it is added to the plan, but the bookkeeping details are suppressed here.

The algorithm generates the plan by following the

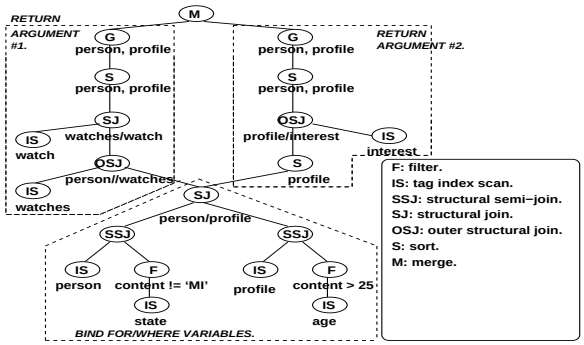


Figure 11: Physical Plan from the GTP of Fig. 2(b).

following stages for each group: (1) compute structural joins; (2) filter based on the evaluable predicates dependent on the contents of more than two pattern nodes if needed; (a predicate is evaluable when all its dependent pattern nodes are bound or the aggregations have been computed) (3) compute value joins if needed; (4) compute aggregation, if needed; (5) filter based on the predicates dependent on the aggregation value, if needed; (6) compute value joins based on aggregation values, if needed; (7) group the return argument, if there is any. Sorting and Duplicate Elimination are added between the stages if needed. Specifically, duplicate elimination is needed in stages 4 and 7.

Example 4 (Translating GTP into a plan) When the above algorithm is applied to the GTP in Fig. 2 we obtain the plan shown in Fig. 11. In this plan, we first do an appropriate sequence of structural joins to find matches for group 0 nodes in the GTP. Two important points to note here are: (1) We rely on a techniques such as [11] to find an optimal order of structural joins, (2) We use structural semi-joins where appropriate so a need for explicit projection and duplicate elimination is avoided [1]. As an example, the structural join between person and state elements is done as a structural semi-join, so even if there are multiple state elements below a person, with value != 'MI', that person would be retained only once. In Fig. 11, bottom, we can see the plan for obtaining the said witness trees. The left operand of the SJ node computes persons with a state != 'MI' while the right operand computes profiles with age > 25. The SJ operator computes (person, profile) pairs satisfying a pc relationship.

Second, we make use of selection conditions in the where clause to restrict generation of bindings for return arguments. E.g., for the first return argument, it is sufficient to find watch subelements for those person elements $p satisfying $p//state != 'MI' and $p/profile/age > 25. This is depicted in Fig. 11 by forking the result of the SJ node above to the two (independent) subplans computing the two return arguments.

Third, rather than compute bindings for the for/where variables and for each return argument separately and combine them with left-outer-join, we use an outer version of structural join. E.g., the left-outer structural join between person and watches under the ad relationship finds all person elements without descendant watches as well as (person, watches) ad pairs.

The output (person, profile) pairs of the SJ node needs to be sorted by profile (node id) before it can be used for outer structural join with interest. Finally, the sequences from the subplans for the two arguments are both sorted by person node id so they can be merged to form the output sequence. ■

4.2 Efficient Implementation

In relational databases, conjunctions of selection conditions are often evaluated through intersection of rid sets, obtained from indices, without accessing the actual data. However, for the most part, query evaluation does process the actual data in the evaluation pipeline. In the case of XML trees, it is possible to encode the tree structure so that quite complex operations can be performed without accessing the actual data itself. On the flip side, the actual data itself is a well-circumscribed tuple in the case of a relational database. But for an XML element, we may be interested in the attributes of this element itself or in its child sub-elements. As such, it is important to distinguish between identification of a tree node (XML element), by means of a node identifier, and access to data associated with this node. This enables us to work with intermediate results that are only partially materialized, and delay data materialization (thus avoiding the cost) until necessary.

Given a heterogeneous set of trees, TPQs use tree pattern matches to identify nodes of interest. In an algebraic expression, it is frequently the case that multiple operators use exactly the same tree pattern. It is computationally profligate to re-evaluate the tree pattern each time for each operator. Instead, we permit the results of a tree pattern evaluation to persist, and thus share with many of the subsequent operators. Pattern tree reuse is akin to common sub-expression elimination. Sometimes, subsequent operators may not use the exact same pattern tree, but rather may use a variation of it. In our implementation we can apply additional conditions to the node-structures known to satisfy the original tree pattern match, as well as extend the tree to include new branches.

XML queries must maintain document ordering. Hence when a join is specified in the query, a nested loops algorithm must be used in order to maintain order. In our implementation we assign node ids based on the document order of each node. If in the document element A precedes B, then node A will have a lower node id[5]. This technique allows us to sort any sequence of trees based on the node id of the root

[5]The same holds for element A containing B

```
Algorithm planGen
Input: GTP G
Output: a physical plan to evaluate G
let GRPs=getGroupEvalOrder(G);
foreach group g in GRPs do {
    let GB=getGroupBasis(g);
    let SJs=the set of structural joins (edges) in g;
    if (g ends with 0)
        let $n=the node related to fv1 in g;
    findOrder(SJs,$n);
    foreach sj in SJs do{
        if (one input stream of sj depends on a node in other
            group) set sj to structural outer join;
        if (one input stream will not be used further)
            project out the unused node and turn sj
                to structural semi-join, if possible; }
    let C={p | p is a predicate in GTP's formula and
        p refers to a node in g and p is evaluable and
        p has not been evaluated }
    add Filter to the plan, which takes the formula from C as its
        argument and the output of SJs as input stream;
    while (∃ predicate p in the join formula and
        p refers to a node in g and p is evaluable
        and p has not been evaluated){
        let JC=the set of such ps that depend on the same two
        inputs; add VJ to the plan, which takes the formula
            from JC as its argument;
        if (∃p ∈ JC && p refers to a node in other group){
            set VJ to outer join;
            make the output of preceding step be VJ's right input
            stream; } }
    let AG={agg($n) | $n in g and agg($n) in GTP's formula}
    add Groupby to the plan, which takes GB and appropiate
        aggregations as its argument;
    let AC={p | p is a predicate in GTP's formula and
        p refers to a node in AG and p is evaluable and
        p has not been evaluated }
    add Filter to the plan, which takes the formula from AC as its
        argument and the output of preceding step as input stream;
    while (∃ predicate p in the join formula and
        p refers to a node in AG and p is evaluable
        and p has not been evaluated){
        let AJC=the set of such ps that depends on the same two
        input; add VJ to the plan, which takes the formula
            from AJC as its argument to the plan;
        if (∃p ∈ JC && p refers to a node in other group){
            set VJ to outer join;
            make the output of preceding step VJ's right input
            stream; } }
    if (g has a return argument)
        add Groupby to the plan, which takes GB as its argument;
    if (g is the last group in its hierarchy)
        add Merge operator to the plan; }
end of Algorithm
```

Figure 12: Algorithm `planGen`

and re-establish document order. Sorting on node id is cheap, as all the information needed is 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 achieves better performance and scalability without sacrificing document ordering.

5 Schema-Aware Optimization

XML, with its optional and repeated elements and irregular structure, poses a great challenge for efficient query processing. In the absence of schema knowledge, we must anticipate all these possibilities for every element! Often XML documents conform to a DTD or XML schema, knowledge of which can benefit in two ways: (i) at a logical level, we can simplify the GTP by eliminating nodes, thus reducing the number of structural joins required; (ii) we can eliminate additional operators (e.g., sorting, duplicate elimination, etc.) in the generated physical plan.

5.1 Logical Optimization

We have identified several types of simplifications of a GTP based on schema information. We discuss just two of these types here. The examples readily generalize.

(1) Internal node elimination: Suppose there are three nodes in a "chain" corresponding to tags a, b, c in a GTP, where b is an ad-child of a and is an ad-parent of c, and b has no other children, has no other local predicates, and does not correspond to a return argument. Then we can remove b from the GTP and make c an ad-child of a, if the schema implies every path from an a to a c passes through a b. The resulting ad edge ($\$a \rightarrow \$b$) is solid iff each of the edges ($\$a \rightarrow \$b$) and ($\$b \rightarrow \$c$) is. E.g., for `book//publisher//address`, suppose the schema implies all address subelements under book must be subelements of publisher. Then we can remove publisher from the GTP. We call this type of schema constraint an *avoidance constraint* since it says b cannot be avoided on a path from a to c. Variations include situations where one or more of the edges could be pc. **(2) Identifying two nodes with the same tag**: E.g., for the query `for $b in .../book, $r in .../review where $b/title = $r/title return <x> {$b/title} {$b/year}</x>`, the corresponding GTP would have two nodes corresponding to title, one in the for-group and the other in the group for return argument 1. The latter can be eliminated and the former can be treated as a return node in addition to its role in the for group, provided the schema says every book has at most one title child. In general, (pc or ad, and solid or dotted) two or more children with tag b of a node of tag a can be identified if the schema implies no node of tag a has more than one child (or descendant) of tag b. In an actual XML database system, the choice of which rewrite rule to use should be a cost-based decision.

5.2 Physical Optimization

We have identified three important sources of physical optimization. All examples below refer to Fig. 11. **(1) Elimination of sorting**: Suppose we want to perform, say an ancestor-descendant structural join on two input streams ordered respectively by person and profile node id's. The algorithm can create the output in either person order or profile order, but in general, not in both. If we choose the former order, it can be used for processing of return argument 1, without further resorting. But for argument 2, where we need to match profiles with child interests, we need to resort the previous output by profile node id's. However, if the schema implies no person can have person descendants, then the output of the strucrual join ordered

by person node id will also be in profile node id order. Conversely, if the schema implies no profile can have profile descendants, then the output ordered by profile order will also be in person order. **(2) Elimination of group-by**: In general, for each return argument, we must group together all elements associated with a given match for the for variables, e.g., for watch and interest in Fig. 11. But if the schema says each profile has at most one interest subelement, then the grouping on the second return argument can be eliminated. For elimination of group-by on the first, the schema needs to imply each person has at most one watches descendant *and* each watches has at most one watch child. **(3) Elimination of duplicate elimination**: In general, for each return argument connected to a for variable by a path of length 2 or more and containing only ad edges, we potentially need a duplicate elimination. E.g., this would be the case for watch element (node $t), if the corresponding expression was `$p//watches//watch` instead of `$p//watches/watch`. Then $t is connected to the for variable $p by the all ad-path ($p→$w→$t). If watches can have watches descendants, then for a given person node, a descendant watch node may be generated multiple times, warranting duplicate elimination. However, if the schema implies watches cannot have watches descendants, this is unnecessary. For this optimization, we need that for each intermediate node x (i.e., excluding the endpoints) on the all ad-path, the schema implies t cannot have t descendants, where t is the tag of x.

5.3 Constraint Inference

We have identified the following kinds of constraints as relevant to GTP simplification: child and descendant constraints and avoidance constraints. They do not exhaust all possibilities but are simple and fundamental. We have developed efficient algorithms for inferring these constraints from a schema specification, such as DTD or XML schema (see [16]). Our algorithms make use of the abstraction of regular tree grammars as structural abstractions of schemas and work off this representation. For brevity, we omit the details. We can show our algorithms are complete and are polynomial time in the size of the regular tree grammar.

5.3.1 GTP Simplification

In this section, we give an algorithm for simplifying a GTP given a set of child, descendant, and avoidance constraints.

The *pruneGTP*(G) algorithm simplifies the GTP G based on the child/descendant constraints and avoidance constraints, typically precomputed from the schema specification. It applies the constraints in the following order, whenever possible: (1) detect emptiness of (sub)queries, (2) identify nodes with same tag, (3) eliminate redundant leaves, and (4) eliminate redundant internal nodes. For brevity, we only show steps (2)-(4) in the algorithm presented.

```
algorithm pruneGTP
Inputs: GTP G < T, F >
Output: a simplified G
while (∃$n1, $n2 in G s.t. $n1.tag = x&&$n2.tag = x ∈ F) {
    if ($n1 and $n2 are siblings && both hold the proper
        (child/decendant) identified constraint with their parent)
        unify($n1,$n2);
    if ($n1's parent is $n2's ancestor (or vice versa) && the
        descendant identified constraint with $n1's parent holds)
        unify($n1,$n2); }
while (∃$n s.t. $n is a leaf of G and $n.tag = x is
    the only predicate about $n in F and $n is not related
    to any fv or lv or return argument and y ↓1/+ x
    where y is $n's parent's tag) {
    delete $n from G; }
while (exists$na, $nband$nc in G s.t. $na is
    $nb's parent and $nb is $nc's parent and $nb.tag = b
    is the only predicate about $nb and the appropriate avoidance
    constraint among $na, $nb and $nc holds) {
    delete $nb; }
end of algorithm

procedure unify
Inputs: two pattern nodes $n1 and $n2, GTP G < T, F >
Outputs: G simplified by combining $n1 and $n2 together
let g1=$n1's group, g2=$n2's group;
make all $n1's decendants be $n2's decendants;
replace $n1 with $n2 in F;
relate to $n2 all fvs, lvs and return arguments related to $n1;
set the group number of all nodes in g1 and g2 to min(g1, g2);
delete $n1;
end of procedure
```

Figure 13: Algorithm `pruneGTP`

We can show the following result on our GTP simplification algorithm:

Theorem 1 (Optimality) *Let C be a set of child and descendant constraints (resp., avoidance constraints). Let G be a GTP. Then there is a unique GTP H_{min} equivalent to G under the presence of C, that has the smallest size among all equivalent GTPs, over databases satisfying C. The GTP simplification algorithm will correctly simplify G to H_{min} and in time polynomial in the size of G.*

When C consists of *both* descendant and avoidance constraints, the minimal equivalent GTP is no longer unique. To see this, consider a simple TPQ (which *is* a GTP!) P corresponding to the XPath expression `t1[.//t2//t3]`. Suppose C consists of the descendant constraint "every `t2` has one or more descendant `t3`'s" and the avoidance constraint "every `t3` that is a descendant of a `t1` is a descendant of a `t2`, itself a descendant of the `t1`". Then P is equivalent to each of the TPQs P_1 = `t1[.//t2]` and P_2 = `t1[.//t3]`, but *not* to `t1`. Both P_1 and P_2 are minimal. In this case, the simplified GTP found by our algorithm would be P_1, since descendant constraints are applied before avoidance constraints. At this time, it is open whether there are smaller GTPs, equivalent to the given GTP under both descendant and avoidance constraints, than found by the algorithm.

6 Experiments

In this section we present the results of experiments demonstrating the value of the GTP. All the experiments were executed using the TIMBER [13] native XML database.

For our data set we used the XMark [24] generated documents. Factor 1 produces an XML document that occupies 479MB when stored in the database. Experiments were executed on a PIII-M 866MHz machine running Windows 2000 pro. TIMBER was set up to use a 100MB buffer pool. All numbers reported are the average of the combined user and system CPU times over five executions[6].

6.1 Navigational and Base Plans

NAV: To compare GTP we 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. We found that the navigational approach is highly dependent on the path size and on the number of children of each node. The smaller the path and the lower the number of children the better the algorithm behaves. For example, XMark query 5 (XM5) has a path site/closed-auctions/closed-auction which corresponds to 1/6/many elements. All of the closed-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.

BASE: Besides the navigational plan, we wanted to use a straightforward tree pattern translation approach that utilizes set-at-a-time processing. We call this approach baseline plan. The XQuery query is first translated into a sequence of TPQs by following the schematic shown in Fig. 4, where each TPQ is represented by a TAX operator taking a tree pattern as its argument. The TPQs in that figure are as follows. P_1 is the TPQ in Fig. 1(a), P_2 is identical to P_1, except conditions on the content of node $\$s$ and $\$g$ are dropped. P_3 is P_2 with $\$s$ and $\$n$ dropped, while P_4 is the TPQ in Fig. 1(b). P_5 is P_4 with the double edge from $\$p$ to $\$w$ replaced by a single edge, and P_6 is shown in Fig. 3(a). P_7 is the TPQ corresponding to the path $\$p \rightarrow \$l \rightarrow \$i$ in Fig. 2(b), except the edge from $\$l$ to $\$i$ is turned into a solid one and all conditions on nodes other than $\$p$, $\$l$, $\$i$ are dropped. P_8 is also shown in Fig. 3(b). The baseline plan is obtained from the TPQs by mapping each edge in each tree pattern to a structural join and mapping each TAX operator to a corresponding TIMBER physical algebra operator. E.g., the TAX join operator mapped to the value join operator in Section 4.1 (see also Section 4.2). Unlike GTP, the baseline plan does not make use of tree pattern reuse.

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

```
Qa: FOR $b IN document("auction.xml")/site
        /open-auctions/open-auction
    RETURN {$b/bidder/increase[./=39.00]/text()}
Qb: FOR $p IN document("auction.xml")/site/people/person
    WHERE SOME $i IN $p/profile/interest
        SATISFIES $i/@category="category28"
    RETURN {$p/name/text()}
Qc: FOR $p IN document("auction.xml")/site/people/person
    WHERE EVERY $i IN $p/profile/interest
        SATISFIES $i/@category!="category28"
    RETURN {$p/name/text()}
```

Figure 14: Queries Qa, Qb, Qc

6.2 Interesting Cases

We executed dozens of queries: those described in the XMark benchmark [24] as well as our own. In our tests we wanted to check the effect of path length, number of return arguments, query selectivity and data materialization cost in general. We selected a few queries that demonstrate the use of these factors. The XQueries mentioned in this section (XM8, XM13 etc) are the corresponding XMark queries. We had to create a few queries, Fig. 14: Qb and Qc to demonstrate quantification, since no XMark query does, and Qa to show a query with with relatively long path and 1 argument in the return clause.

We used an index on element tag name for all the queries, which given a tag name, returns the node ids. We used a value index, which given a content value, returns the node ids, to check the condition on content for queries XM5, XM20, Qa, Qb and Qc. Results are summarized in Fig. 15 (ignore column SCH, which we will discuss in Section 6.4).

GTP outperforms NAV and BASE for every query tested, sometimes by one or two orders of magnitude. All algorithms are affected by the path length. NAV is affected the most, since it will have to do more costly iterations to find the answer. GTP and BASE are both affected by the increased cost of more structural joins, but not as much as NAV.

Query selectivity does not affect all algorithms. NAV will do the same number of iteration even if zero results are produced. GTP and BASE are affected by query selectivity in the form of paying for the extra data materialization cost to produce the answer. So we noticed that the speedup of GTP over NAV and over BASE decreases if more results are to be produced.

The number of return arguments also affects the algorithms. NAV is not affected much, as it has to do the same number of iterations anyway. GTP is affected in terms of data materialization costs and having to do more sorts and groupings to get the final result. BASE is affected the same way as GTP plus the extra cost of having to do the new tree pattern matches. However, the speedup of GTP over BASE increases, since GTP does a tree pattern extension for every argument in the return clause.

In general, data materialization cost affects both GTP and BASE. NAV is not affected much since it

Query	Tested Algorithm				Query Description
	NAV	BASE	GTP	SCH	
XM5	10.77	0.89	0.20	0.05	1 argument/return, short path, value index
Qa	77.93	8.92	0.47	0.08	1 argument/return, long path, value index
XM20	25.90	11.83	1.09	0.50	> 1 argument/return, med path, value index
Qb	42.46	23.41	0.39	0.37	> 1 arg/return, quantifier some, high selectivity, value index
Qc	42.52	25.63	1.09	1.05	> 1 arg/return, quantifier every, low selectivity, value index
XM13	50.45	1.90	0.50	0.48	> 1 arg/return, long path
XM19	128.10	70.03	29.49	28.12	> 1 arg/return, lots of generated results
XM8	108.92	111.45	15.66	15.07	> 1 argument/return, single value join, nested
XM9	159.11	180.32	20.50	18.82	> 1 argument/return, multi value join, nested

Figure 15: CPU timings (secs) for XMark factor 1. Algorithms used: NAV = Navigational plan, BASE = Base plan, GTP = GTP algorithm, SCH = GTP with schema optimization. The queries are XMark XQueries (XM5, XM20, ...) and the queries (Qa, Qb, Qc) seen in figure 14.

Query	XMark scale factor				
	0.05	0.1	0.5	1	5
XM13	0.02	0.05	0.25	0.50	2.43
XM8	0.58	1.15	7.88	15.66	73.52

Figure 16: CPU timings (secs). Using GTP with no optimization or value index.

has already paid the cost of getting all children.[7] Notice that in queries with joins or with lots of results the data materialization cost makes BASE perform poorly and reduces the speedup of GTP over NAV. Note that GTP is the only algorithm that *could* benefit from an index on the join value and perform very well in queries with joins. Unfortunately such an index was *not* available in our tests. So GTP performance speedup over NAV decreases when data materialization cost is very high. BASE sometimes performs even worse then NAV on queries with joins BASE materializes data "early" and then has to carry the penalty of this materialization in all the joins and tree pattern matches.

6.3 Scalability

We tested queries XM13 and XM8 for scalability. We used XMark factors 0.05(24MB), 0.1(47MB), 0.5(239MB), 1(479MB) and 5(2387MB). XM13 is a simple selection and XM8 is a nested FLWR query that includes a join. As we can see in Fig. 16, GTP scales linearly with the size of the database.

6.4 Schema-Aware Optimization

The column SCH in Fig. 15 shows the performance of GTP after schema-aware optimization. We see that schema knowledge can greatly enhance performance in some cases, but helps very little in others. Schema-aware optimization performs well when (result) data materialization is not the dominating cost. We also note that when the path is of the form 1/1/1/few and schema optimization converts it to 1//few then the benefit is again small. Schema-aware optimization performs well when the path is of the form many/many/many and is converted to many//many. We present in Fig. 17 a comparison between GTP and schema-aware plans, using queries

[7] In TIMBER, nodes are clustered with their children. So the disk cost of getting all children id's is almost the same as getting all children id's and their values.

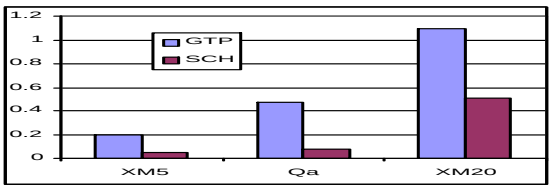


Figure 17: CPU timings (sec). Comparison of GTP and GTP with schema optimization plans.

XM5, Qa, and XM20. It shows schema-aware optimization produces much faster executions in these cases.

7 Related Work

There are three major approaches to XML data management and query evaluation. Galax [21] is a well-known example of a navigation-based XQuery query processing system. Relational approaches to XQuery implementation include [10, 22, 8, 25, 20], while [3] uses an object-relational approach. Some examples of native approaches to XML query processing include Natix [9], Tamino [19], and TIMBER [13].

Most previous work on native XQuery implementation has focused on efficient evaluation of XPath expressions via structural join [2] and holistic join [7], and optimal ordering of structural joins for pattern matching [11]. TIMBER [13] makes extensive use of structural joins for pattern match, as does the . Niagara system [17]. We are not aware of any papers focusing on optimization and plan generation for XQuery queries as a whole for native systems. The closest is the "dynamic intervals" paper [8], which is based on translating queries into SQL. They make use of the well-known interval encoding of trees and then assign these intervals dynamically for intermediate results. A direct experimental comparison with their approach is difficult since their tests were run on their own homegrown XQuery processor, whereas our algorithms were implemented and tested in the TIMBER native XML system. Even though the computer on which our tests were run is slower than reported by [8], for an XMark factor of 1, for queries XM8, XM9, XM13 for which we did not use any value indices like them, we observed our response was 3-20 times faster. This is admittedly an ad hoc comparison, but it does

give a general indication.

Recently, there has been much interest in optimizing (fragments of) XPath expressions by reasoning with TPQs or variants thereof, possibly making use of available schema knowledge [15, 23, 18]. GTPs enable similar logical optimization to be performed for XQueries as a whole, with or without schema knowledge.

8 Summary and Future Work

This paper has taken a significant step towards the efficient evaluation of XQuery. We proposed a novel structure called a generalized tree pattern that summarizes all relevant information in an XQuery into a pattern consisting of one or more trees. GTPs can be used as a basis for physical plan generation and also as a basis for logical and physical query optimization, exploiting any available schema knowledge. We demonstrated the effectiveness of GTPs with an extensive set of tests comparing GTP plans with plans directly generated from XQuery as well as with alternate navigational plans. In most cases, GTP plans win by at least an order of magnitude. We intend to implement GTP-based plan generation as an integral part of the TIMBER system.

GTPs provide an elegant framework with which to study query containment for XQuery, to our knowledge for the first time. This is significant, since we expect this will be applicable also for query answering using (XQuery) views and for incremental view maintenance. We presented an algorithm for schema-based simplification of GTPs and hence XQuery. More work is needed to fully exploit all schema knowledge and comprehensively calibrate its performance benefits.

Whereas our experimentation has been limited to the TIMBER system, and hence can directly be extrapolated only to native XML database systems, the GTP concept is equally applicable to relational mappings of XML. A rigorous evaluation of the benefits GTPs bring to relational XML systems remains part of our future work.

References

[1] S. Al-Khalifa and H. V. Jagadish. Multi-level Operator Combination in XML Query Processing. pp. 286–297, CIKM 2002.

[2] S. Al-Khalifa et al. Structural joins. A primitive for efficient XML query pattern matching. pp. ICDE 2002.

[3] K. Runapongsa and J.M. Patel. Storing and Querying XML Data in Object-Relational DBMSs. pp. 266-285, EDBT Workshop XMLDM 2002.

[4] A. Berglund et al. XML path language (XPath) 2.0. `http://www.w3.org/TR/xpath20/`, Nov. 2002.

[5] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes. W3C Recommendation. http://www.w3.org/TR/xmlschema-2/, May 2001.

[6] S. Boag et al. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery`, Nov. 2002.

[7] N. Bruno et al. Holistic twig joins: Optimal XML pattern matching. pp. 310-321, SIGMOD 2002.

[8] D. DeHaan et al. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding, SIGMOD 2003. To appear.

[9] T. Fiebig et al. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.

[10] D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[11] Y. Wu et al. Structural Join Order Selection for XML Query Optimization. ICDE 2003. To appear.

[12] H.V. Jagadish et al. TAX: A Tree Algebra for XML. pp. 149-164, DBPL 2001.

[13] H. V. Jagadish et al. TIMBER: A native XML database. *VLDB Journal*, 11(4):274–291, 2002.

[14] H.V. Jagadish et al. Implementing XQuery using TAX. Tech. Report, U. Michigan. June 2003. In preparation.

[15] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. pp. 65-76, PODS 2002.

[16] M. Murata et al. Taxonomy of XML schema languages using formal language theory. Extreme Markup Languages. Montreal, Canada, August 2001.

[17] J. Naughton et al. The Niagara Internet Query System. http://www.cs.wisc.edu/niagara/papers/ NIAGARAVLDB00.v4.pdf.

[18] F. Neven and T. Schwentick, XPath Containment in the Presence of Disjunction, DTDs, and Variables. pp. 315-329, ICDT 2003.

[19] H. Schoning. Tamino - A DBMS designed for XML. pp. 149-154, ICDE 2001.

[20] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. pp. 302-314, VLDB 1999.

[21] J. Simeon et al. Galax, An open implementation of XQuery. `http://db.bell-labs.com/galax/`.

[22] I. Tatarinov et al. Storing and querying ordered XML using a relational database system. pp. 204-215, SIGMOD 2002.

[23] P.T. Wood. Containment for XPath Fragments under DTD Constraints, pp. 300-314, ICDT 2003.

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

[25] X. Zhang et al. Honey, I Shrunk the XQuery! – An XML Algebra Optimization Approach. pp. 15-22, WIDM 2002.

[26] C. Zhang et al. On supporting containment queries in relational database management systems. SIGMOD 2001.