Multi-level Operator Combination in XML Query Processing

Shurug Al-Khalifa and H. V. Jagadish [*]
University of Michigan
Ann Arbor, MI 48109, U.S.A.
{shurug,jag}@eecs.umich.edu

ABSTRACT

A core set of efficient access methods is central to the development of any database system. In the context of an XML database, there has been considerable effort devoted to defining a good set of primitive operators and inventing efficient access methods for each individual operator. These primitive operators have been defined either at the macro-level (using a "pattern tree" to specify a selection, for example) or at the micro-level (using multiple explicit containment joins to instantiate a single XPath expression).

In this paper we argue that it is valuable to consider operations at each level. We do this through a study of operator merging: the development of a new access method to implement a combination of two or more primitive operators. It is frequently the case that access methods for merged operators are superior to a pipelined execution of separate access methods for each operator. We show operator merging to be valuable at both the micro-level and the macro-level. Furthermore, we show that the corresponding merged operators are hard to reason with at the other level.

Specifically, we consider the influence of projections and set operations on pattern-based selections and containment joins. We show, through both analysis and extensive experimentation, the benefits of considering these operations all together. Even though our experimental verification is only with a native XML database, we have reason to believe that our results apply equally to RDBMS-based XML query engines.

Categories and Subject Descriptors:
H.2.2[Database Management]:Physical Design—Access Methods.

General Terms: Design, Performance.

Additional Key Words and Phrases: XML, Query Processing.

[*]Supported in part by NSF under grant 115-0208852 and DM1 0075447.

1. INTRODUCTION

It is well-recognized that set-oriented data processing is essential for good performance in any data management system. XML data is no exception. Towards this end, there has been considerable recent work towards developing a bulk algebra for XML query, and efficient access methods for operators in this algebra. (See Sec 2). Much of the work along these lines is applicable whether the XML database is implemented natively or on a relational engine.

Unfortunately, there is not yet universal agreement on an algebra for XML query evaluation. We argue in this paper that there may be good reason for this. In Sec. 2, we will look at past work on this subject, and establish the background necessary for this paper. There are two main alternatives with regard to set-oriented XML manipulation. The first approach manipulates sets of trees directly. The operators in such an algebra are heavyweight, but more directly expressive of user intent. A core operation is a "pattern tree match" selection (that is, given a set of documents (XML trees) find all occurrences of a specified tree pattern in any of these documents). The second approach is to have a lower level algebra that manipulates sets of elements (nodes in trees). The operators here more directly reflect the implementation. In fact a single pattern tree match selection operator can itself be computed as a sequence of containment (or structural) joins. We call these two, respectively, *macro-level* and *micro-level* algebras (and operators).

In both macro-level and micro-level algebras, the basic operators considered correspond to "intuitive" unit operations such as selections, projections, joins, and set operations. Quite naturally, the focus in developing efficient access methods has been restricted to these unit operations. In this paper we explore the benefits that can be obtained by pushing in projections and set operations. Realizing these benefits requires a new class of composite access methods that evaluate these composite operators in one swoop.

We show that operator merging is valuable for both macro-level operators (Section 3) and micro-level operators (Section 4). We develop rewrite rules for pushing projections and set operations into structural pattern match selection in Section 3. We also develop new access methods for the various merged operators. Specifically, in Section 4, we present access methods for projection merged with Containment Join and negation (Set Difference) merged with Containment Join.

We then present a brief analytical assessment in Section 4.3 and an extensive experimental evaluation,in Section 5. These sections show the benefit to be obtained from the new composite operators for a wide variety of data sets and query types.

A significant consequence of the work presented here is a substantial expansion of the class of access methods considered for XML query processing. More importantly, there is a strong case made for working with separate algebras at two levels: to obtain maximum benefit it appears that XML query processing requires both a macro-algebra and a micro-algebra! We conclude with this and other implications of our work in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 Query Algebra and Operators

There is no shortage of algebras for data manipulation. Ever since Codd's seminal paper there have been efforts to extend relational algebra in one direction or another. Even in the XML context, several algebras have been proposed. [3] is an influential early work that has impacted XML schema specification. The W3C working group on XML Query has recently issued an algebra document [8]. The focus of this algebra is to provide a formal semantics for XQuery [6]. It is not suitable for set-oriented processing.

Set-oriented algebras for XML can be divided in two main classes – those that deal with trees and those that deal with tuples/elements. We consider each in turn.

Tree Algebras and Macro Operators

XML documents are tree-structured. Therefore it is appropriate to treat an XML database as a collection of trees. An algebra for processing XML queries should therefore comprise queries that map one or more collections of trees to collections of trees. Queries in most query languages proposed for XML, and XQuery in particular, typically specify patterns of selection predicates on multiple elements that have some specified structural relationships. (These structural selections may be specified either through XPath expressions in the FOR clause of an XQuery expression or as logical predicates in the WHERE clause, or both). Such a structural pattern match can be considered a "selection" operator in a *macro-level* algebra. The result is a set of trees, one tree corresponding to each set of bindings that satisfies the given predicates and structural relationships.

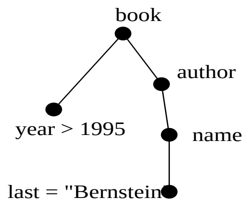
Ideas along these lines have been incorporated into an object-oriented database, and an algebra developed for these in the Aqua project [15]. The focus of this algebra is the identification of pattern matches, and their rewriting, in the style of grammar production rules. These ideas are also the basis for TAX [12] a tree algebra for XML, that is being used as the basis for the implementation of the Timber native XML database[16]. There is also a grammar-based algebra, shown equivalent to a calculus, for manipulating tree-structured data using production rules [10].

EXAMPLE 2.1. *Fig. 1 shows a simple XQuery expression and its corresponding pattern tree. We are seeking titles of books in the database that have an author with a last name of* Bernstein *and have an associated* year *after 1995.*

A macro-level algebra would implement this entire expression as a single pattern-tree based selection operator (to select matching books) followed by a projection operator (to return their titles). (The title *is a sub-element rather than an attribute of* book*. Strictly speaking, we would perform a projection on* book *and then follow pointers from each book in the result to output its title.)* ♢

```
FOR $b IN document("db.xml")
//book[author/name/last
= "Bernstein"]
WHERE $b//year > 1995
RETURN $b/title
```

(a) An XQuery expression



(b) Its Pattern Tree

Figure 1: A simple example

Tuple Algebras and Micro-Level Operators

Tree pattern selections are unlikely to be implemented atomically. Most implementations, whether in a native XML database or on top of a relational engine, will break down a tree pattern selection into a sequence of simple (single-node) selections and "containment joins" that capture the required structural relationships. A *micro-level* algebra can be defined for this purpose.

EXAMPLE 2.2. *A micro-algebra would break up the selection pattern of Fig. 1 into one selection operator per node (e.g.* tag= "book" *or* (tag= "year") && (content > 1995)*) and one containment join operator per edge (e.g. a containment join of these two node lists will satisfy the left-most edge in the query pattern, and find books with a year greater than 1995). The result of the sequence of joins would then be projected on the* book *element, after which its* title *can be obtained navigationally as before.* ♢

In [7], the authors present an algebra for XML, defined as an extension to relational algebra, that is practical and implemented. A "bind" operator is used to create sets of (tuples of) bindings for specified labelled nodes. Similarly, [13] describes a navigational algebra for querying XML, treating individual nodes as the unit of manipulation, rather than whole trees. The algebra in [9], used as the basis for the Niagara[17] XML data management system, is also in the same category.

2.2 Query Processing Implementation

The key access method of concern for a macro-algebra is one for structural pattern matching. This task is complex enough that it is performed in three (conceptual) stages:

1. Identify lists of candidate elements in the database to match each node in the specified structural pattern. These lists are obtained through evaluation of predicates local to these nodes, using indices where available.

2. Find combinations of candidate elements, one from each list, that satisfy the required structural relationships. These combinations are usually built up one structural relationship at a time. The choice of order is a critical determinant of performance.

3. Apply any conditions that involve multiple nodes in the structural pattern to eliminate some combinations.

This access plan can be expressed as a sequence of physical micro-algebra operators, and looks somewhat like a relational join plan where local selections are applied first, the join is computed, and additional global conditions can be checked in a final step. In short, the actual implementations are likely to be similar whether the query optimization is carried out in a micro-algebra or a macro-algebra.

A central operation in all cases is the *containment join*. Given two sets of elements U and V, a containment join returns pairs of elements (u, v) such that $u \in U$, $v \in V$, and u "contains" v (that is, node u is an ancestor of node v in the tree representation of the appropriate document). A containment join is asymmetric, and we will refer to one node (u) as the *ancestor node* and the the other node (v) as the *descendant node*. (E.g. in Fig. 1, there are four containment joins, one corresponding to each edge. For the leftmost edge, the ancestor node is the "book" node and the descendant node is the "year" node.) A special case of the containment join is the *immediate containment join*, where we require that node u be a direct parent (rather than any ancestor) of node v. This special case could have performance implications, but no new conceptual issues, so we will not separately mention such joins for the bulk of our paper.

A simple node numbering scheme[17, 1, 18, 5] is commonly used to avoid computing transitive closures of inclusion relationships to determine a containment join. This scheme associates a pair of numbers (start, end) with each node (element) in the tree (XML document) where these represent the word offset of the element start tag and end tag respectively in the document. In other words, each element is transformed into an integer interval. Element u contains element v if and only if $\mathsf{start}(u) < \mathsf{start}(v)$ and $\mathsf{end}(u) > \mathsf{end}(v)$. One can keep lists of elements sorted by their start value where possible, to make such containment joins easier to compute. When we say "sorted by u" in the sequel, we'll mean "sorted by $\mathsf{start}(u)$".

Containment Join Implementation

As we saw above, containment joins are central to XML query processing, and structural pattern matching in particular. A typical containment join has a (simple) selection predicate at each end, applied locally to the node in question. Indices are likely to be available to help evaluate these selection predicates (at least partially, if the predicate has multiple clauses). We have three main options in evaluating such a join: we could scan the entire database, we could use an (the most selective) index to find candidate nodes for one end of the join and then navigate from there (following parent/child pointers), or we could use indices to find candidates for both ends of the join and then compute a containment join between these candidate sets. Previous work [19, 1, 5] has shown conclusively that the last of these options is almost always the best.

While the specific choice of join algorithm used for containment join is orthogonal to the issues we wish to explore in this paper, to make matters concrete we will restrict our description (and our experimentation) to the class of algorithms shown to be the best in [1].

3. MACRO OPERATOR MERGING

We consider several operators that could follow a pattern tree selection: projection, intersection, union, and set difference. For each of these four operators, we consider the benefit of merging it with the pattern tree selection(s) that it follows. At the macro level, merging operators seems intuitive and simple. But since both the micro level operators that compose the selection and the mapping between the macro selection and the micro containment joins will get affected, the merging process is no longer as simple.

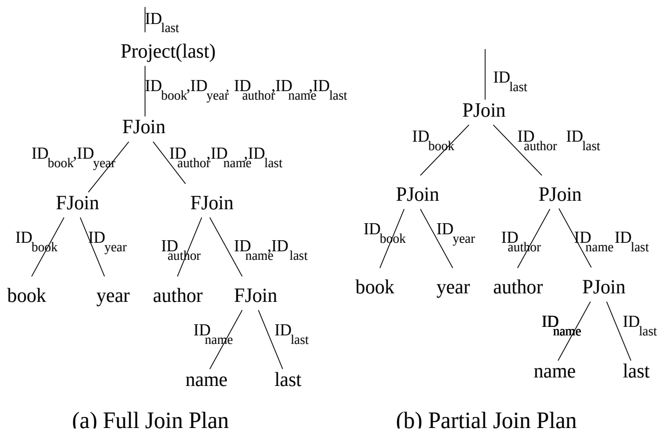


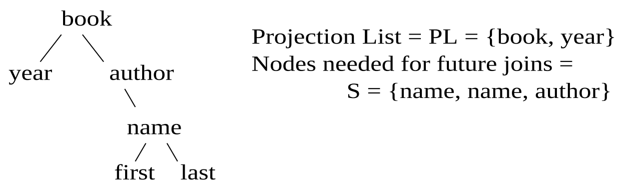
Figure 2: The difference between Full and Partial containment joins.

3.1 Projection Merging

EXAMPLE 3.1. *Consider pattern tree presented in Fig. 1. The query seeks books with author last name "Bernstein" and year greater than 1995. We will alter it a bit to return* last. *Fig. 2a is a plan that can be used to evaluate the query. To evaluate this query, as described above, we would first generate lists of candidate nodes that match individual nodes in the pattern (e.g. "book" nodes, "author" nodes, and so on). Then we will compute a sequence of containment joins, one for each edge in the pattern. For instance, suppose the first join is* name-last. *The result of this join is a set of* pairs *of nodes that jointly satisfy the relevant portion of the pattern. The next containment join, say between* author *and* name *actually joins a set of* name-last *pairs with a set of* author *pairs to produce a set of* author-name-last *triples. Finally, after all containment joins have been evaluated, we have a set of 5-tuples, that are the result of the pattern match selection. A projection operator is then used to focus on the* last *nodes and eliminate the others.* $\diamondsuit$

What we discussed in the above example is what we call a sequence of *full* containment joins (FJoin). Each of these retain both inputs tuples. We argue the need for a projection-aware select operator. Like the select operator, this operator requires a pattern tree as a parameter, and finds sub-trees in each input data tree that match the pattern tree. But, in addition, this operator comes with a projection list comprising references to pattern tree nodes. Only nodes referenced in this list are retained in the output. The rest are ignored. The projection-aware select operator, just like the ordinary select operator, is evaluated by decomposing it into a series of containment joins. Fig. 2b is a plan that performs project-aware selection. Each binary containment join is aware of what needs to be retained (last in our example.) Instead of passing the two inputs' tuples, it passes only the ones to be projected (or needed in a later join). We call this type of join, *partial* containment join. Formally, we can define:

DEFINITION 3.1. **Full Containment Binary Join:** *A* full containment binary join *is as an operator* $FJoin : \mathcal{E}^m \times$


book
year
author
name
first
last
Projection List = PL = {book, year}
Nodes needed for future joins =
S = {name, name, author}

Input vectors to partial joins	output vector (input to next join)	Set of nodes not in PL but need to be kept for future joins
name PJoin first $\longrightarrow$	name	S = {name, author}
name PJoin last $\longrightarrow$	name	S = {author}
name PJoin author $\longrightarrow$	author	S = {}
author PJoin book $\longrightarrow$	book	S = {}
book FJoin year $\longrightarrow$	book,year	S = {}

Figure 3: Execution of a sequence (one of many possible) of partial binary containment joins to evaluate a specified pattern tree selection.

$\mathcal{E}^n \times [1 \ldots m] \times [1 \ldots n] \rightarrow \mathcal{E}^{(m+n)}$. *Given inputs* $(x_1, \ldots, x_m)$, $(y_1, \ldots, y_n)$, *i, and j, the output is* $(x_1, \ldots, x_m, y_1, \ldots, y_n)$, *the concatenation of the first two inputs, provided that* x_i *is an ancestor of* y_j, *and is empty otherwise.* $\diamondsuit$

Here, $\mathcal{E}$ is the set of all XML elements in the database. $\mathcal{E}^m$ is a vector of m elements, representing some tree fragment. x_1 is the ID of a node that has participated in a previous join. And so are x_2 through x_m. In Fig. 2a, in the topmost FJoin, x_1 corresponds to IDs of tuples satisfying book and x_2 corresponds to IDs of tuples satisfying year. For the same FJoin, y_1 corresponds to IDs of tuples satisfying author, y_2 corresponds to IDs of tuples satisfying name, and y_3 corresponds to IDs of tuples satisfying last.

DEFINITION 3.2. **Partial Binary Containment Join:** *A* partial binary containment join *is as an operator* $PJoin$: $\mathcal{E}^m \times \mathcal{E}^n \times [1 \ldots m] \times [1 \ldots n] \times \{0,1\}^{m+n} \rightarrow u\mathcal{E}*$. *Given inputs* $(x_1, \ldots, x_m)$, $(y_1, \ldots, y_n)$, *i, j, PL, the output is* $\{z_k | (z_k \in (x_1, \ldots, x_m) || z_i \in (y_1, \ldots, y_n)) \&\& (k \in PL)\}$, *provided that* x_i *is an ancestor of* y_j, *and is empty otherwise.* $\diamondsuit$

The projection list, PL, is formally recorded as an $m+n$-long bit vector, with each bit indicating whether the corresponding node is retained in the projection. $\mathcal{E}*$ is a vector of an undetermined number of elements (but this number is exactly the number of 1s in PL and hence is no larger than $m+n$). In other words, the output is that for the full binary containment join, $(x_1, \ldots, x_m, y_1, \ldots, y_n)$, projected down to elements in PL, the projection list. Basically, $PJoin$ acts in the same exact way as $FJoin$ does, except that it does not concatenate the input vectors blindly. Instead, it outputs a new vector with only nodes that are referred to in the PL. In Fig. 2b, the topmost PJoin outputs only IDs of last because it is the only one in the projection list.

DEFINITION 3.3. **Projection Minimal:** *An expression is said to be* projection-minimal *if every projection is applied as early as possible. That is, every intermediate result in the expression has all unnecessary elements projected out.* $\diamondsuit$

THEOREM 3.1. *Let* Q' *be the expression obtained after rewrite rule (4) has been applied as many times as possible recursively to a given project-select expression* Q. *Then,* Q' *is projection minimal and is equivalent to* Q. $\diamondsuit$

We devise a simple algorithm to push projections in, based on the above theorem and the observation that each edge in the pattern tree represents exactly one containment join. Therefore each node in a pattern must participate in exactly as many containment joins as there are edges incident on it. To keep track of this, we create a multi-set S comprising each node in the pattern tree repeated as many times as it has edges, minus 1. To save space, we can leave out projection list nodes from S altogether. A node u is retained in the result of a binary containment join if it is in the final projection list or is included in S. Otherwise, u is projected out. Every time a node is used in a join, remove one occurrence of u from S.

EXAMPLE 3.2. *Fig. 3 presents a pattern to be matched (an extension of the pattern tree in Fig. 1). PL, the projection list, is specified to be* book *and* year. *The set S has nodes* name *in it twice because it has three edges incident.* book *is not in S since it is in PL. The other three (leaf) nodes have only one edge incident each. A possible sequence of partial binary containment joins to evaluate this query is shown, along with the manipulations of the set S. In the first step,* name *is joined with* first. first *is not retained in the result since it is neither in S nor in PL.* name *is retained since it is in S, but one occurrence of* name *is removed from S. In the last step,* book *and* year *are retained in the result since both are in PL.* $\diamondsuit$

3.2 Set Operations

In the relational world, union compatibility is an important consideration with respect to set operations. In XML, since heterogeneous collections are allowed, this is not an issue. (One consequence is likely to be that set operations are more prevalent in XML query processing than in relational query processing – though we do not have enough knowledge of XML query processing to verify this hypothesis.) Set intersection is really the same thing as a conjunctive condition. So, no new operators or access methods are required. Unlike intersection, set difference cannot be expressed directly as part of a pattern tree selection. However, a new access method that merges containment join and difference, we call it *negated containment join*, turns out to be very useful, as we shall shortly see.

3.2.1 Set Union

The standard technique to perform set operations is to sort the two inputs and then merge them. An appropriate choice of query plan can avoid the need to sort the lists. Specifically, we would like each input set to be sorted by the root node of each element tree.

In the relational context, union distributes selection over the same relation as follows:
$\sigma_p R \cup \sigma_q R \; = \; \sigma_{p \vee q} R$ where p, q are selection predicates. Given the more complex pattern-tree selections we have to deal with, we must find the equivalent rule for merging two pattern trees. We observe the following equivalence:
Given two pattern trees, PT_1 and PT_2, let PT_c be a common component of the two pattern trees such that (i) $PT_1 - PT_c = PT_1'$ and $PT_2 - PT_c = PT_2'$ are both also trees, and (ii) Node i in PT_c has node j in PT_1' such that edge (i,j) is in PT_1, if and only if node i also has some node k in PT_2' such that edge (i,k) is in PT_2.

LEMMA 3.1. *There exists exactly one such node i provided that* PT_c, PT_1', *and* PT_2' *are each non-empty.* $\diamondsuit$

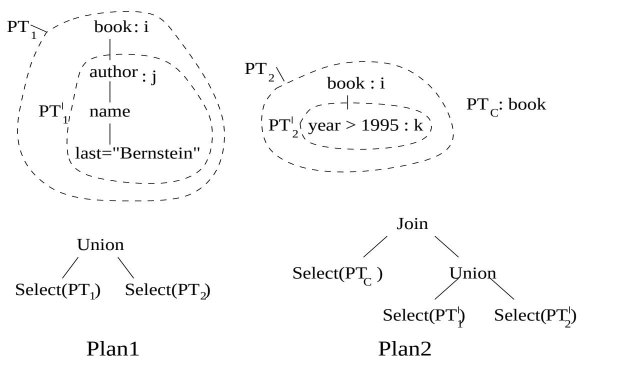


Figure 4: Different pattern trees and plans involved in evaluating a query that asks for books with author last name of "Bernstein" OR year $>$ 1995

The lemma follows from the definition of a tree. If PT_1 is a tree and is decomposed into two trees, then the two component trees have exactly one edge between them. Using this lemma, we can develop a rewriting rule:

$$Select_{PT_1}(D) \cup Select_{PT_2}(D) \; = \; UJoin(Select_{PT_c}(D), Select_{PT_1'}(D), Select_{PT_2'}(D), i, j, k)$$

All variables in this rule are defined as above. $UJoin$ is a new *Union Containment Join* operator. It is similar to $FJoin$. It differs in that it takes two candidate descendant tree sets (the second and third arguments) and correspondingly two descendant node identifiers (the last two arguments), rather than just one each in $FJoin$. Consider a variant of our running example in Fig 4, where we seek books with author last name of "Bernstein" OR year $>$ 1995. This is the union of two pattern match selections, one for each disjunct (PT_1 and PT_2 in Fig. 4). PT_c is the common part of the two patterns, the root node book in this case. This is also node marked i. PT_1' is the remainder of the first pattern, author, name, and last nodes, with the first of these being the k node. An ij edge "connects" the two patterns. Similarly, PT_2' comprises the single node year. This is the node marked k.

A common case where this occurs is when j and k are each the root for the corresponding sub-pattern, as in our running example. We thus have a choice of unioning books that satisfy either disjunct (plan1 in Fig. 4), or unioning the disjunctive condition and then joining only once with book (plan2 in Fig. 4).

$$Select_{PT_1}(D) \cup Select_{PT_2}(D) \; = \; FJoin(Select_{PT_c}(D), Select_{PT_1'}(D) \cup Select_{PT_2'}(D), i, j)$$

4. MICRO-OPERATOR MERGING: NEW ACCESS METHODS

At the macro-level, we considered a pattern tree selection as a single (heavy-weight) operator in the previous section, and discussed benefits that this operator could derive from operators that follow. An alternative, micro-level algebra approach is to break up a pattern tree selection into multiple containment join operators. In this section, we will show that merging operators that follow is a good idea even for containment joins. Not only is this true for the projection and set operators we have discussed before, but in addition it is frequently worthwhile to merge multiple binary containment join operators and evaluate a single multi-way containment join instead. The new multi-way containment join is very useful especially for long chains. However, bushy multi-way containment joins (for twig patterns) turn out to be not a good idea. For lack of space, we don't include experiments and discussions of the multi-way containment join in this paper. For further detail, refer to the full-length version of the paper found in [2]. In the following subsections, we touch upon highlights of the algorithms. At the micro level, it is impossible to reason with operators at the macro level because operators at the micro level are not aware of the macro level. Instead, they are used by the macro level operators to perform the different operations.

4.1 Partial Binary Containment Join

In this section, we discuss modifications made to the original algorithms presented in [1] to achieve the partial binary containment joins.

We start with the simplest case, the descendant-sorted containment join. Since we wish to retain only the descendant nodes in the output, we need not join the descendant with all stack elements. If the stack has at least one (ancestor) element in it, the descendant is output immediately. We can additionally get rid of the stack altogether for an ancestor-descendant join (vs. a parent-child join), merely retaining the single cell at the bottom of the stack. If this cell is occupied when a candidate descendant node is considered, then the descendant node is output, and otherwise it is not.

Turning to a binary containment join with only ancestor nodes retained, several optimizations are possible. The greatest benefit is that no lists need to be kept since we do not need to output the descendant. Furthermore, all nodes on the stack can be reported (in bottom-to-top order) as soon as one descendant is found. After that one can skip all descendants that might have joined with the popped elements.

In the general case, we may retain all, some or none of the elements in the previously joined lists on either side of the containment join. Retaining elements from one side does not affect any of the techniques presented above. If we retain elements from both sides of the containment join, we do not get the significant "semijoin-like" algorithmic benefits discussed in this section. However, we still benefit due to reduction in the size of the output.

4.2 Negated Containment Join

A negated binary containment join is a primitive operator in the evaluation of set difference. The algorithm for this access method is almost identical to that for the ancestor-projected partial binary join operation. When a matching descendant node is compared with the top of the stack, all candidate ancestor nodes on the stack are popped, as before. However, these nodes are not output; instead they are rejected. Any candidate ancestor nodes that survive until popped from stack due to the start cursor having gone too far forward (a new node has a start key greater than the end key of surviving ancestors on stack) are the ones that should be output.

Algorithm	*Cost*	*Notes*
Descendant-sorted full binary join	$v_1 * n_1 \ + \ v_2 * n_2 \ + \ (v_1 + v_2) * n_{12}$	$n_{12} \leq n_1 * n_2$
Ancestor-sorted full binary join	$v_1 * n_1 \ + \ v_2 * n_2 \ + \ (v_1 + v_2) * n_{12} * (1 + 2f_1)$	ancestor node is 1
Descendant-sorted partial binary join	$v_1 * n_1 \ + \ v_2 * n_2 \ + \ u * m$	$u \leq v_1 + v_2$, $m \leq n_{12}$
Ancestor-sorted partial binary join	$v_1 * n_1 \ + \ v_2 * n_2 \ + \ u * m * (1 + 2f_1)$	$u \leq v_1 + v_2$, $m \leq n_{12}$
Negated binary containment join	$v_1 * n_1 \ + \ n_2 \ + \ v_1 * m * (1 + 2f_1)$	$m \leq n_1$

Table 1: I/O Cost for Various Merged Containment Join Operations

4.3 Analysis

To understand the effect of the various algorithms suggested above, we construct a cost model using a number of simplifying assumptions for tractability. We measure data in the units of "elements", ignoring differences in size between elements. Each "element" unit corresponds to some number of bytes or some fraction of a disk page. A summary of our analysis results is presented in Table 1. In this table, n_i is the cardinality of the i^{th} input, and v_i is its "tuple"-size (number of elements already joined). n_{ij} (or m) is the cardinality of the output, and u is the number of elements retained in the output. f_i is a *nesting factor*, which takes a value between 0 and 1, and is used to indicate how "recursively nested" node i is in the given data set. Let $\mathcal{I}$ represent the set of all nodes in the database that satisfy the predicates to match with node i in the containment join at hand. f_i is the fraction of nodes in $\mathcal{I}$ that have a descendant in $\mathcal{I}$. Typical data sets have low nesting factors for most element types. A nesting factor of 0 is called the *no overlap* property. For instance, a book cannot be a descendant of a book.

5. EXPERIMENTS

We ran an extensive set of experiments on a wide range of real and synthetic data. We expect, based on the analysis in the previous section, that operator merging is a good idea for relational implementations as well as native implementations. Also, we expect it to be beneficial on operators at both micro and macro levels. Another level of operators we experimented with include ones that span both micro and macro levels. These operators have the ability of being pushed into selection either at the macro level or micro level. Projection and set difference can be applied after evaluating parts of the pattern tree (macro level). Also, they can be applied at the micro level by actually using the partial containment join for the projection and the negated containment join for the set difference.

Our experimental results are limited to Timber [16], a native XML database we are building, not just because we had this one database easily available, but additionally because good XML query processing requires appropriate access method support in relational engines, and the requirements for these are only now being discovered through research, so they have not yet made their way into commercial products that we could use.

Our code is implemented in Visual C++. All experiments were run on a Windows NT 550 MHz machine with 256 MB of RAM. Each experiment was run five times. The least and greatest values were ignored and the average of the middle three was taken.

We show the impact of operator merging at both macro and micro levels on a wide variety of data sets. These include both real data (parts of Sigmod Record and DBLP) and synthetic data created with the IBM XML generator [11] using "real" DTDs, including several obtained from [17], and the popular Organization DTD obtained from AT&T.

5.1 Macro Level Experiments

In this section, we present experiments with pushing into selections operators that can only be pushed at the macro level. This includes set union and intersection. In Table 2a we present the results of evaluating a set operation after tree pattern selections. For the same tree pattern selections, we consider the evaluation of set union and set intersection of the results. In each case, we present timing numbers with the set operation computed afterwards versus the set operation pushed in. We find in all cases that pushing in set operations is beneficial.

We also altered the pattern tree structure used in different queries. In Table 2b, we considered four different pattern tree structures when applying set union and intersection. For each pattern, we measured time when the set operation is and is not pushed into the selection. The first pattern tree is the simple pair query consisting of an ancestor and a descendant. In this case, the savings are not big. But the more complex the query gets, the more savings we achieve. Chain1 in the table corresponds to a 3-node chain with low nesting of root. Chain2 is similar to chain1 except that the root has high nesting (a root node can have a descendant root node). The query structure experiments were performed on the organization data set because of its high nesting of different elements which allows diversity in query structure.

5.2 Combined Micro and Macro Level Experiments

In this section, we evaluate the effect of pushing operators that can be either pushed at the macro or micro level. The operators are projection and set difference. When pushing these operators in selections, we have the option of pushing at the macro level (not using new containment joins) or at the micro level (using new joins that replace full binary containment joins).

First, we will consider pushing projections into selections. In tables 3 and 4, we present experiments run on multiple data sets. In each case, the pattern comprised a chain of three nodes. We ran two queries: with the selection requiring that the middle node in the chain be an immediate child of the root and the leaf node be an immediate child of the middle node (corresponding to a single "/" in Xpath), and with this requirement not being imposed (corresponding to a double "//" in Xpath). The two queries are referred to respectively as "parent-child" and "ancestor-descendant". We present the results of evaluating a projection after the three-node pattern selection versus pushing it in to occur after a pair in the pattern has been evaluated and then after the final join (macro level) versus pushing it in to occur along

Data Set	*Union*			*Intersection*		
	no push	*push*	*%*	*no push*	*pushed*	*%*
SIGMOD	34.72	24.79	(71)	26.41	21.09	(80)
DBLP	21.30	16.02	(75)	17.38	14.77	(85)
Club	30.41	20.52	(68)	26.48	20.32	(77)
Bib.	31.45	28.96	(92)	12.59	12.08	(96)
Actors	95.71	71.66	(75)	87.54	71.66	(82)
Movies	35.53	23.89	(67)	30.68	23.47	(76)
Personnel	57.29	55.69	(97)	31.88	31.37	(98)
Org.	188.07	158.88	(85)	105.16	60.76	(58)

(a)

Query	*Union*			*Intersection*		
Struct.	*no push*	*pushed*	*%*	*no push*	*pushed*	*%*
Pair	16.76	16.30	(97)	7.88	8.24	(104)
Twig	1111.93	573.87	(52)	10.95	9.53	(87)
Chain1	102.20	89.88	(88)	15.49	12.28	(79)
Chain2	188.07	158.88	(85)	105.16	60.76	(58)

(b)

Table 2: (a)Time (in seconds) measuring the effect of pushing set union and intersection (macro level) into selections on multiple real and synthetic data sets (b)Effect of pushing set union and intersection on different query structures

Data Set	*Parent Child Join*				*Ancs Desc Join*			
	not pushed	*macro push*	*micro push*	*%*	*not pushed*	*macro push*	*micro push*	*%*
SIGMOD	20.10	18.21	16.32	(81)	15.22	13.98	12.54	(82)
DBLP	31.98	31.09	29.78	(93)	33.56	32.21	30.92	(92)
Club	23.22	20.23	19.12	(82)	21.21	18.04	15.24	(72)
Bibliography	41.10	38.69	35.21	(86)	40.02	38.22	33.29	(83)
Actors	43.13	40.52	37.01	(86)	43.12	40.10	35.99	(83)
Movies	18.23	17.98	16.44	(90)	20.98	17.93	16.09	(77)
Personnel	61.88	58.12	55.30	(89)	55.13	51.18	46.12	(84)
Organization	49.10	47.12	45.10	(92)	50.14	47.10	45.34	(90)

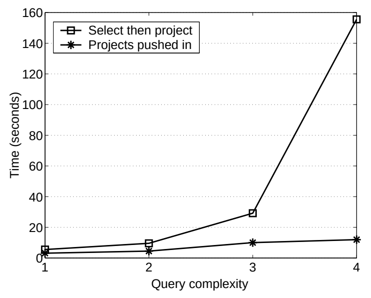
Table 3: Time (in seconds) measuring the effect of pushing projections into selections (macro and micro levels) on multiple real and synthetic data sets when projecting descendant.

Data Set	*Parent Child Join*				*Ancs Desc Join*			
	not pushed	*macro push*	*micro push*	*%*	*not pushed*	*macro push*	*micro push*	*%*
SIGMOD	57.52	52.51	47.98	(83)	19.56	15.12	12.56	(64)
DBLP	47.98	46.43	39.78	(83)	46.12	42.53	38.56	(84)
Club	20.11	17.20	13.87	(69)	19.88	16.32	12.45	(63)
Bibliography	60.55	53.20	41.32	(68)	58.10	41.89	37.11	(64)
Actors	25.85	21.10	17.55	(68)	25.31	21.09	15.32	(61)
Movies	71.04	55.83	48.98	(69)	73.12	59.20	44.45	(61)
Personnel	63.19	55.84	45.01	(71)	67.85	58.04	43.11	(64)
Organization	59.12	58.29	54.85	(93)	289.23	72.22	38.21	(13)

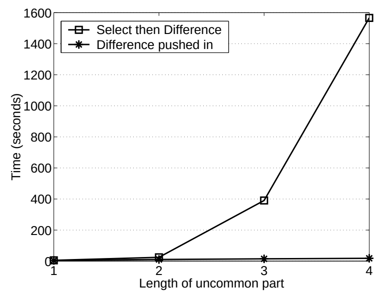
Table 4: Time (in seconds) measuring the effect of pushing projections into selections (macro and micro levels) on multiple real and synthetic data sets when projecting ancestor.

Data Set	*not pushed*	*macro push*	*micro push*	*%*
SIGMOD	33.07	19.85	11.69	(35)
DBLP	30.58	21.73	7.08	(23)
Club	161.35	108.00	89.12	(55)
Bibliography	108.49	78.10	65.20	(60)
Actors	25.43	10.13	0.06	(0.24)
Movies	28.55	25.99	13.47	(47)
Personnel	57.39	42.55	8.52	(15)
Organization	390.50	308.57	14.65	(4)

Table 5: Time (in seconds) measuring the effect of pushing set difference into selections (macro and micro levels) on multiple real and synthetic data sets



(a)



(b)

Figure 5: (a) Varying query complexity when pushing projections in (b) varying number of common nodes when pushing set difference in

with the containment join (micro level). We do this for both projection of the leaf node (table 3) and of the root node (table 4) in the chain pattern tree. We find that in all cases pushing in projections is a good idea. The benefit is typically small in the case of leaf node projection, but quite significant in the case of root node projection.

When pushing in operators, the more complex the query, the greater the benefit – we find ancestor-descendant queries uniformly benefiting more from the projection push in than the corresponding parent-child queries. Among the data sets, the one where the greatest benefit was observed was the organization data set, which involved the greatest degree of nesting, leading to complex evaluation.

Now, we will consider pushing set difference into selection. In table 5, we present experiments run on multiple data sets. In each case, we computed the set difference between a pair pattern and a 3-node chain pattern with the top two nodes are identical to those of the pair pattern. A query of the type "get all departments that have employees who do not have phone numbers". This will convert to a department-employee pattern on the left side of the difference operator and a department-employee-phone on the right side of the operator. This was a description of the no pushing case. In the macro push case, the root (department) of both pattern trees was pulled out, and the difference was evaluated on employee and employee-phone. The department then is joined with difference output. In the micro push case, the 3-node pattern is evaluated with replacing the join between the bottom two nodes with the negated containment join. The percentage column in the table is between the no push case and the micro push case. In all cases, we benefit the most from pushing difference all the way in.

Finally, we considered varying query complexity when pushing projections in and varying number of common nodes in query when pushing difference in. The results are in figure 5 a and b. Since the macro push case has always been inferior to the micro push case, in both figures we ignore the macro push case and present a comparison between the two extreme cases: the no push and the micro push.

6. CONCLUSION

XML query processing has been modelled in terms of macro-algebras (which operate on entire trees) and micro-algebras (which operate on individual nodes). In this paper, we have explored the relative ease of certain optimizations in one versus the other, and shown that there can be substantially differences between the two. In both cases, we have shown that it is often very valuable to merge operators, and have a single access method evaluate a combination of operators.

The contributions of this paper include the development of an optimization framework that exploits the duality between macro-algebras and micro-algebras for XML; the development of new access methods for operator combinations, including a projection containment join and a negation containment join; an analytical assessment of the benefits of the new access methods compared to their unmerged originals; and an extensive experimental evaluation of these benefits, with a variety of data sets, a variety of queries, and the variation of various operating conditions.

A significant consequence of our work is that it is not enough to consider XML query optimization purely at the micro-algebra or purely at the macro-algebra level, with simple algebraic operators. Instead, one has to consider access methods for combinations of operators, switching between the micro and macro levels as needed.

7. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. Patel, D. Srivastava and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of ICDE*, 2002.

[2] S. Al-Khalifa and H. V. Jagadish. Combining Operators in XML Query Processing. *University of Michigan technical report.* Available at http://www.eecs.umich.edu/db/timber/

[3] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for XML. W3C XML Query Working Group Note, Sep. 1999.

[4] C. Beeri and Y. Tzaban. SAL: An algebra for Semi-Structured Data and XML. *ACM SIGMOD Workshop on the Web and Databases*, pp. 37–42, Philadelphia, PA, June 1999.

[5] N. Bruno, D. Srivastava, and N. Koudas. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of SIGMOD*, 2002.

[6] S. Boag, D. D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simon and M. Stefanescu. XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3.org/TR/xquery/, December 20, 2001.

[7] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. SIGMOD*, pages 141–152, 2000.

[8] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. W3C Working Draft. June 7, 2001.

[9] Leonidas Galanis, Efstratios Viglas, David J. DeWitt, Jeffrey. F. Naughton, and David Maier. Following the Paths of XML Data: An Algebraic Framework for XML Query Evaluation. 2001. Available at http://www.cs.wisc.edu/niagra/papers/algebra.pdf.

[10] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. In *Proc. ACM SIGMOD*, pages 263–272, 1989.

[11] IBM. XML Generator available from http://www.alphaworks.ibm.com/tech/xmlgenerator

[12] H. V. Jagadish, L. V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Proc. of Intl. Workshop on Databases and Programming Languages*, Marino, Italy, Sep. 2001.

[13] B. Ludascher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven evaluation of virtual mediated views. In *Proc. EDBT*, pp. 150–165, 2000.

[14] A. Sahuguet. Kweelt. Available from http://db.cis.upenn.edu/Kweelt/.

[15] B. Subramanian, T. W. Leung, S. L. Vandenberg, S. B. Zdonik. The AQUA approach to querying lists and trees in object-oriented databases. In *Proc. ICDE*, 1995.

[16] U. of Michigan. The Timber system. http://www.eecs.umich.edu/db/timber/.

[17] U. of Wisconsin. The Niagara system. http://www.cs.wisc.edu/niagara/.

[18] Y. Wu, J. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proc. of EDBT*, 2002.

[19] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the ACM SIGMOD Conference on Management of Data*, 2001.