Cover Page

Contact Author

	Name	Shurug A. Al-Khalifa
	Email	shurug@eecs.umich.edu
	Addr	1913 Lindsay Lane, Ann Arbor, MI 48104, U.S.A
	Tel	(734) 677 2668

Paper Info

	Ref. No.	562
	Topic Area	Core Database Technology
	Category	Research
	Title	Combining Operators in XML Query Processing
	Authors	Shurug A. Al-Khalifa and H. V. Jagadish
	Topics relevant to the paper	Semi-structured Data, XML

Combining Operators 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.

In this paper we devise new access methods to implement various combinations of two or more such operators merged together. Specifically, we consider the influence of projections and set operations on pattern-based selections and containment joins. We also study multi-way containment joins. We show, through both analysis and extensive experimentation, that access methods for merged operators are frequently far superior to a pipelined execution of separate access methods for each operator.

Our results are applicable whether one considers primitive operators 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). 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.

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.

Quite naturally, the basic operators considered in such efforts correspond to "intuitive" unit operations such as selections, projections, joins, and set operations. Quite naturally, too, the focus in developing efficient access methods has been restricted to a subset of these unit operations – particularly selections and joins, since these are often applied early in a query plan and frequently dominate the computational effort required. The main idea we explore in this paper is that subsequent operations, particularly projections and set operations, may profitably be "pushed in" and performed at the same time as the early selections and joins. Similarly, multiple joins can sometimes benefit from being executed together (i.e. multi-way joins can be substantially faster than a sequence of binary joins). Realizing these benefits requires a new class of composite access methods that evaluate these composite operators in one swoop. Of course, the concept of merging operators has been used effectively in relational databases for years. Most commercial relational engines implement semi-joins more efficiently than full natural joins, for instance. In XML, we'll see that the savings are greater, not only due to duplicate elimination and smaller intermediate results, but also due to the inherent efficiency of the merged operation compared to its constituent parts, for instance on account of not having to examine some parts of the input.

With XML, a complication is that there is no universal agreement on an algebra for XML query evaluation. 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 op-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

erators 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).

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.
- Negation (Set Difference) merged with Containment Join, and
- Multiway Containment Join (merging two or more binary containment joins).

We then present a brief analytical assessment in Section 4.4 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. 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.

In the context of the Web, we should mention Hy+ [8] and the models for semi-structured data (see, e.g., Lore [15] and UnQL [5]); all propose query languages, with more or less effort at an accompanying algebra. Even in the XML context, several algebras have been proposed. [2] is an influential early work that has impacted XML schema specification. The W3C working group on XML Query has recently issued an algebra document [9]. The focus of this algebra is to provide a formal semantics for XQuery [6]. It is not suitable for set-oriented processing.

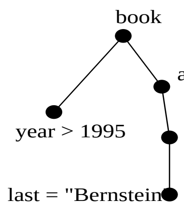
Set-oriented algebras for XML can be divided in two main classes – those that deal with trees and those that deal with tuples/elements. We consider each in turn below.

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.

```
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

Ideas along these lines have been incorporated into an object-oriented database, and an algebra developed for these in the Aqua project [17]. 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 [13] a tree algebra for XML, that is being used as the basis for the implementation of the Timber native XML database[18]. There is also a grammar-based algebra, shown equivalent to a calculus, for manipulating tree-structured data using production rules [11].

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.) $\diamond$

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. $\diamondsuit$

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, [14] describes a navigational algebra for querying XML, treating individual nodes as the unit of manipulation, rather than whole trees. The algebra in [10], used as the basis for the Niagara[19] 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[19, 1, 20, 4] 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 [21, 1, 4] 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]. The basic idea is to exploit the tree structure of XML – a depth first pre-order traversal of the tree will, at some point in the traversal, place every contained node (descendant) on top of every containing node (ancestor) in the stack. Rather than traverse every node in the tree, one could restrict the traversal to just the candidate ancestor and candidate descendant nodes (which potentially satisfy the respective local selection conditions, as identified by indices). Thus a single pass over the (sorted by start position) lists of candidate ancestor and descendant nodes suffices to compute the containment join.

The order in which outputs are produced is usually material in query processing. The simple stack-based join suggested above generates output sorted by the start position of the descendant node in the containment join. In [1], a variant of this algorithm is also presented, where the computed results are saved and output sorted by the start position of the ancestor node. A clever list management technique permits this saving and sorting with very little memory and at most one write to and read back from disk. These algorithms serve as the starting point for the new access methods we develop in Section 4.

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.

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. ♢

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

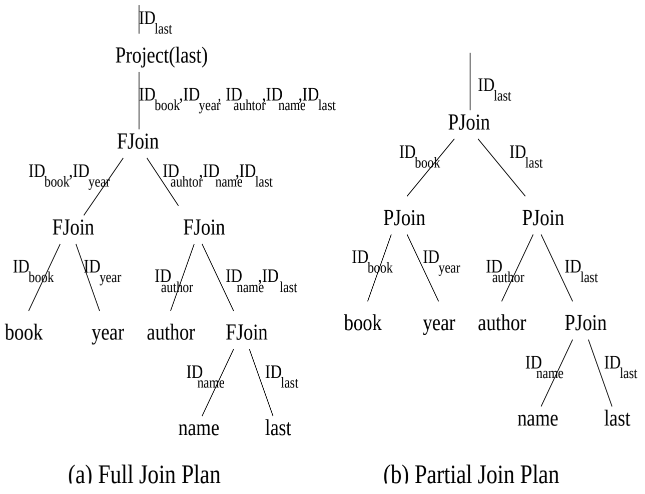


Figure 2: The difference between Full and Partial containment joins.

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 $FJoin : \mathcal{E}^m \times \mathcal{E}^n \times [1 \ldots m] \times [1 \ldots n] \rightarrow \mathcal{E}^{(m+n)}$. 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. ♢

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.

Given a pattern tree selection operation, based on a pattern PT, let $PT_j = PT - PT_i$, where PT_j includes the node j, j is the parent of node i in PT, and i is root of PT_i.

$$\begin{aligned} &Select_{PT}(D) = \\ &\qquad FJoin(Select_{PT_j}(D), Select_{PT_i}(D), j, i) \end{aligned} \tag{1}$$

Thus, a selection on pattern tree PT is expressed in terms of selections on simpler pattern tree PT_i and PT_j. This rule is applied recursively until all pattern trees are reduced to single nodes, at which point standard selection techniques can be used.

When there is a projection immediately following the selection operation, the rewrite rule can be modified to push the projection in, as follows:

$$\begin{aligned} &Project_{PL}(Select_{PT}(D)) = \\ &= Project_{PL}(FJoin(Select_{PT_j}(D), \\ &\qquad\qquad Select_{PT_i}(D), j, i)) \qquad (2) \\ &= Project_{PL}(FJoin(Project_{PL\cup\{j\}}(Select_{PT_j}(D)), \\ &\qquad Project_{PL\cup\{i\}}(Select_{PT_i}(D)), j, i)) \qquad (3) \\ &= PJoin(Project_{PL\cup\{j\}}(Select_{PT_j}(D)), \\ &\qquad Project_{PL\cup\{i\}}(Select_{PT_i}(D)), j, i, PL)) \qquad (4) \end{aligned}$$

This rule is also applied recursively until all pattern trees are reduced to single nodes. Once the base single node selections are evaluated, all the remaining operations are repeated executions of $PJoin$. (Projection pattern lists may mention nodes not in their operands, but these make no computational difference, and can easily be removed, if desired.) Note that the projection pattern list is not invariant – as the projections are pushed in, additional information has to be retained to permit future joins to be evaluated correctly.

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. $\diamondsuit$

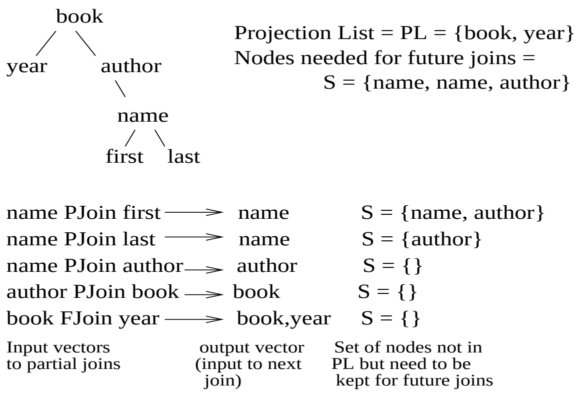

book
year
author
name
first
last
Projection List = PL = {book, year}
Nodes needed for future joins =
S = {name, name, author}
name PJoin first → name S = {name, author}
name PJoin last → name S = {author}
name PJoin author → author S = {}
author PJoin book → book S = {}
book FJoin year → book,year S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

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

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.)

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.* $\diamondsuit$

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:

$$\begin{aligned} &Select_{PT_1}(D) \cup Select_{PT_2}(D) = \\ &\qquad UJoin(Select_{PT_c}(D), \\ &\qquad Select_{PT_1'}(D), Select_{PT_2'}(D), i, j, k) \quad (5) \end{aligned}$$

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.

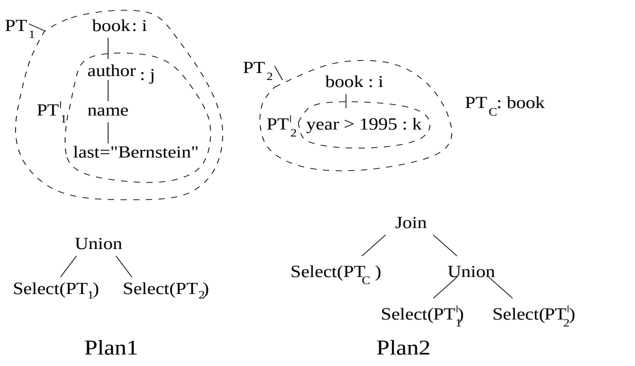


Figure 4: Different pattern trees and plans involved in evaluating a query that asks for books with author last name of "Bernstein" OR year > 1995

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).

$$\begin{aligned} &Select_{PT_1}(D) \cup Select_{PT_2}(D) = \\ &\qquad FJoin(Select_{PT_c}(D), \\ &\qquad Select_{PT_1'}(D) \cup Select_{PT_2'}(D), i, j) \quad (6) \end{aligned}$$

3.2.2 Set Intersection

An intersection is really the same thing as a conjunctive condition, which is what the pattern tree represents. So no new operators or access methods are required. Once the intersection has been pushed in appropriately, it can be replaced by merging the pattern trees corresponding to the selection results being intersected. For instance, it is easy to see that the final intersection query in the example above is the same as the more complex single pattern match of Fig. 1.

3.2.3 Set Difference

Suppose we wish to find books published since 1995 except those with Bernstein as the last name of the author. This is easily expressed as the difference of two selection queries: the first finds books published since 1995 and the second finds books with Bernstein as (last name of) author.

Set difference, unlike intersection, cannot be expressed directly as part of a pattern tree selection. However, an access method to compute set difference merged with containment join, we call it *negated containment join*, turns out to be very useful, as we shall shortly see.

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. For lack of space, we only touch upon highlights of the algorithms here. Greater detail can be found in the appendix.

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.

4.3 Multiway Joins

When there is a sequence of joins in relational databases, it is frequently the case that the join attributes are different for each join. In the case of the multiple containment joins required in a pattern tree, though, the join "attribute" is the position of the element in the document. All candidate matches for all nodes in the pattern tree are sorted by start position for the various binary containment join access methods discussed above. Recent results in this regard are reported in [4].

There are two ways in which patterns can grow – there can be longer chains, or there can be more children at a given node. While a general pattern will require both of these operations to grow from a single node, it is instructive to consider the two extreme cases in isolation. This is what we do next.

Given a chain pattern, the multi-way chain containment join access method will evaluate it in one fell swoop rather than in multiple smaller pieces. Rather than just keep one type of candidate ancestor node on the stack, we keep all nodes on the chain except the bottom-most descendant node. When a descendant candidate node arrives it is compared against nodes in the stack as before. If it matches, then entire chains of matches can be created and output in the specified order.

Now consider a *twig* pattern (two descendants share an ancestor), and an associated multi-way twig containment join access method. When a candidate descendant node arrives, it has to be retained as a partial match with the candidate ancestors on the stack at that time. At the same time, we can create some output based on previous partial matches that have been stored. The output too has to stay in lists, in the manner described above. In this algorithm, unlike the original one, the bottom element in the stack needs to keep lists of descendants too for future matches.

4.4 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 re-

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	$v_1 * n_1 \ + \ n_2 \ + \ v_1 * m * (1 + 2f_1)$	$m \leq n_1$
Chain three-way containment join	$v_1 * n_1 \ + \ v_2 * n_2 \ + \ v_3 * n_3 \ + \ u * m * (1 + 2f_1)$	$u \leq v_1 + v_2 + v_3$, $m \leq n_{123}$
Bushy three-way containment join	$v_1 * n_1 \ + \ v_2 * n_2 \ + \ v_3 * n_3 \ + \ u * m * (1 + 2f_1)$ $+ \ k_2 * v_2 * n_2 \ + \ k_3 * v_3 * n_3$	$u \leq v_1 + v_2 + v_3$, $m \leq n_{123}$

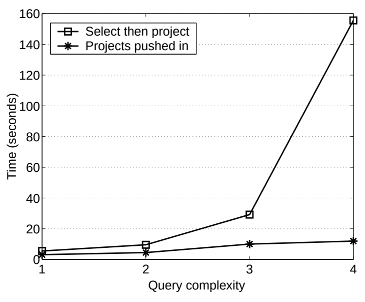
Table 1: I/O Cost for Various Merged Containment Join Operations

Data Set	*Project Descendant*						*Project Ancestor*					
	Parent Child Join			*Ancs Desc Join*			*Parent Child Join*			*Ancs Desc Join*		
	Original	*Proj Pushed*	*%*	*Original*	*Proj Pushed*	*%*	*Original*	*Proj Pushed*	*%*	*Original*	*Proj Pushed*	*%*
SIGMOD	35.77	33.27	(93)	11.70	9.49	(81)	44.73	38.49	(86)	13.70	10.00	(73)
DBLP	24.90	23.10	(93)	25.36	23.34	(92)	32.58	27.95	(85)	31.66	26.26	(83)
Club	11.68	10.43	(89)	12.21	10.84	(89)	14.66	11.06	(75)	13.92	9.60	(96)
Bibliography	39.51	36.87	(94)	39.56	34.61	(88)	46.17	34.97	(76)	44.98	30.34	(68)
Actors	32.06	29.93	(94)	32.21	28.53	(89)	37.58	28.73	(76)	36.80	25.21	(69)
Movies	14.03	13.10	(94)	14.24	12.60	(89)	16.43	12.62	(77)	16.25	11.15	(69)
Personnel	45.13	42.76	(94)	44.73	39.82	(89)	52.53	40.63	(78)	50.32	34.85	(69)
Organization	40.95	39.15	(95)	31.48	22.19	(70)	52.27	46.95	(90)	147.87	22.49	(15)

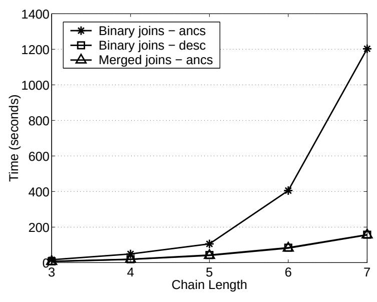
Table 2: Time (in seconds) measuring the effect of pushing projections into selections on multiple real and synthetic data sets

Data Set	*Union*			*Intersection*			*Difference*		
	Original	*Pushed*	*%*	*Original*	*Pushed*	*%*	*Original*	*Pushed*	*%*
SIGMOD	34.72	24.79	(71)	26.41	21.09	(80)	33.07	11.69	(35)
DBLP	21.30	16.02	(75)	17.38	14.77	(85)	30.58	7.08	(23)
Club	30.41	20.52	(68)	26.48	20.32	(77)	31.25	20.41	(65)
Bibliography	31.45	28.96	(92)	12.59	12.08	(96)	108.49	65.20	(60)
Actors	95.71	71.66	(75)	87.54	71.66	(82)	25.43	0.06	(0.24)
Movies	35.53	23.89	(67)	30.68	23.47	(76)	28.55	13.47	(47)
Personnel	57.29	55.69	(97)	31.88	31.37	(98)	57.39	8.52	(15)
Organization	188.07	158.88	(85)	105.16	60.76	(58)	390.50	14.65	(4)

Table 3: Time (in seconds) measuring the effect of pushing set operations into selections on multiple real and synthetic data sets



(a)



(b)

Figure 5: (a) Pushing projections with increasing query complexity (b) Multi-way chain containment joins with increasing chain length

sults is presented in Table 1. A detailed derivation is presented in the appendix. 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. k_i is the fraction of input i that will be considered for output. It appears in the bushy multi-way equation because in the algorithm, lists of input are kept with stack elements as well as an output list.

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. Our experimental results are limited to Timber [18], 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.

5.1 Initial Experiments

The first question to ask is, does operator merging make a difference? In Tables 2 and 3, we show the impact of operator merging 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 [12] using "real" DTDs, including several obtained from [19], and the popular Organization DTD obtained from AT&T.

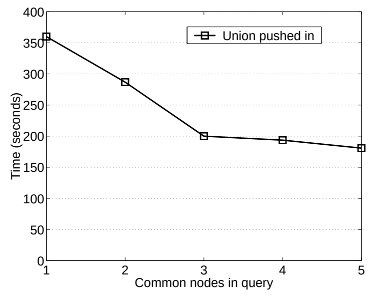
We ran numerous queries over all of the data sets, and report in Table 2 results for a representative set of two-node selection pattern queries. In each case, the pattern comprised an ancestor node and a descendant node. We ran two queries: with the selection requiring that the descendant node be an immediate child of the ancestor 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 two-node pattern selection versus pushing it in to occur along with the containment join. We do this for both projection of the ancestor node and projection of the descendant node. We find that in all cases pushing in projections is a good idea. The benefit is typically small in the case of descendant node projection, but quite significant in the case of ancestor node projection.

In Table 3 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, set intersection, and set difference 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.

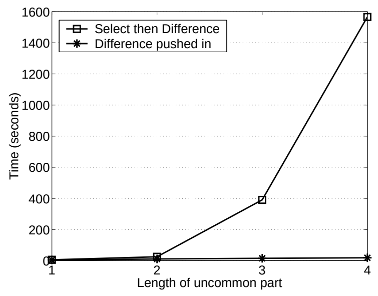
When pushing in operators, the more complex the query, the greater the benefit – we find ancestor-descendant queries uniformly benefitting more from the projection push in than the corresponding parent-child queries. Among the data sets, the one where the greatest benefit was observed was the organization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

To study the benefits of multi-way containment joins, we present results in Table 4a and b for the two extreme cases: a bushy "twig" pattern and a chain pattern. In each case, we focus on simple three-node queries. We require results sorted by the pattern root node in all cases. In a 3-node chain, we compute the lower containment join first, sorted by ancestor, and use the result as the descendant part of the upper containment join. In a 3-node twig, we can compute either containment join first, but must sort by ancestor, and use the result as the ancestor part of the other containment join. We present timing numbers for various queries on several data sets, evaluated as such sequences of two binary joins and also as single multi-way joins.

The conclusion is that chain multi-way joins are always advantageous. The extent of the benefit is greater when there is nesting in the root node of the chain. Twig multi-way joins, though, more often did worse rather than better, compared to the sequence of binary joins. This is for the reasons we predicted in our analysis in Section 4.4. We also ran several experiments with more complicated pattern trees, and saw similar results, though less clearly, because of the confounding of multiple effects. Our conclusion is that containment join merging is beneficial only for chains in the pattern. We focus on chain multi-way joins for the remainder of our study.

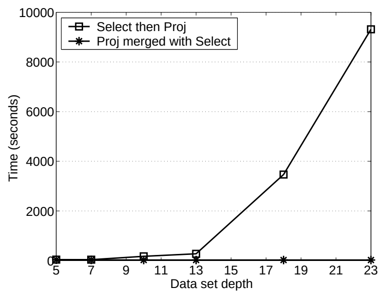


(a) Set Union

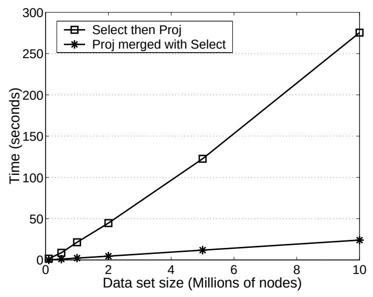


(b) Set Difference

Figure 6: Comparing degrees of set operation push into pattern tree selection



(a) Varying Depth of Data Set



(b) Varying Size of Data Set

Figure 7: Evaluating the effect of pushing in projections as the data set is scaled

Data Set	*Binary Joins*	*Merged*	*%*
SIGMOD	27.37	23.05	(84)
DBLP	18.36	17.72	(97)
Club	145.61	154.57	(106)
Bibliography	397.95	446.71	(112)
Actors	15.07	15.82	(105)
Movies	150.82	184.30	(122)
Personnel	20.34	14.70	(72)
Organization	55.16	67.87	(123)

(a) Twig Pattern

Data Set	*Binary Joins*	*Merged*	*%*
SIGMOD	22.72	17.64	(78)
Club	21.43	17.37	(81)
Bibliography	17.79	15.13	(85)
Actors	52.36	48.77	(93)
Movies	12.32	10.28	(83)
Personnel	19.53	16.36	(84)
Organization	16.26	6.43	(40)

(b) Chain Pattern

Table 4: Performance (in seconds) of Three-way Containment Joins

Query	*Union*			*Intersection*		
Structure	*Orig*	*Push*	*%*	*Orig*	*Push*	*%*
Pair	16.76	16.30	(97)	7.88	8.24	(104)
Twig	1111.93	573.87	(52)	10.95	9.53	(87)
Chain (low nesting)	102.20	89.88	(88)	15.49	12.28	(79)
Chain (high nesting)	188.07	158.88	(85)	105.16	60.76	(58)

(a)

Query	*No Optim.*	*Merged Chain*	*Push Union in*
1	102.197	35.441	89.879
2	188.070	189.734	158.879

(b)

Table 5: (a)Time (in seconds) measuring the effect of query structure in pushing set operations into tree pattern selections (b)Comparing (in seconds) two exclusive optimizations on two different queries

5.2 Query Scaling

In the preceding subsection we hinted at the fact that the greater the complexity of the query evaluation, the greater the benefits of operator merging. Here, we see this result in spades. To be able to present comparative results, this section onwards we present results on a single data set, corresponding to the organization DTD. We chose this data set because of its complexity. Complex queries are not even possible on very simple data sets. For instance, the DBLP data set is only around two levels deep, so we cannot evaluate complex chain queries on it. We did run some of these experiments on other data sets, and observed similar trends.

Fig. 5a shows this effect for projection queries. We kept adding to the query complexity (measured in terms of the number of constituent binary containment joins) and projecting out only one node. Table 5a shows this effect for set union and set intersection queries.

Fig. 5b shows that the benefits of chain multi-way joins increase with the length of the chain, and hence the arity of the multi-way join, increases. This happens because of the reduced amount of disk I/O involved when performing a multi-way chain.

When pushing operators into complex tree pattern selections, we also have the option of pushing in only part way. Fig. 6a shows the result of pushing in a union operator in stages. As the common part pulled out after the union goes up, the benefit of operator merging also goes up. Fig. 6b shows the same effect for pushing in set difference in stages. As the uncommon part goes up, the difference in performance and thus the benefit of pushing in difference increases. This happens with the larger uncommon part because it costs more to evaluate.

5.3 Data Set Scaling

Based on our analysis and previous experiments, we expect that operator merging should provide greater benefits as the size and complexity of the data set grows. To test this hypothesis we generated a number of different synthetic data sets using the same DTD (the organization DTD). The depth of the data set greatly affects the complexity of the query evaluation, so we wanted to study the effect of depth in addition to sheer size.

In all cases, we present results for a simple three-node chain selection followed by a projection on the ancestor node. This query is similar to one that asks for all authors with the last name being "Bernstein". (The three nodes in the selection are author, name, and last. The result is projected on author). Similar trends were observed with other queries, including pair queries.

Figure 7a shows the times for evaluating the same query against several data sets all the same size (10 million nodes) but with varying depth. We see that the cost of evaluating a projection after the selection increases greatly with depth whereas the cost of the merged operator is barely affected. The reason is that there are more ancestor-descendant relationships in a deeper data set, so the size of the pre-projection intermediate result is much larger while the size of the final output is not affected much. The merged operator succeeds in never creating this large result that it has subsequently to pare down.

Fig. 7b shows the time to evaluate the same query, with and without projection pushed in, for a number of data sets of varying size, with the depth kept fixed at 13 levels. As the size of the data set increases, the query takes longer to evaluate, irrespective of the access plan chosen. However, on a relative basis, the merged operator evaluation does better on larger data sets.

5.4 Operator Combinations

Thus far, we have considered one operator merge at a time. In this section, we turn to the issue of applying multiple operator merges concurrently. Fig. 8 shows the effect of pushing projections in and using a multi-way chain containment join. The three curves show respectively the case with just projections pushed in while using binary joins, the case with a multi-way containment join followed by a projection, and the case where the projection has been pushed into a multi-way containment join. We do not show the base case, with no optimizations applied, since this performs substantially worse – plotting this curve would make it hard to see the differences between the three curves of interest. The reader interested in the base case can refer to Fig. 5b which considers the same data set and same queries.

Our conclusion from Fig. 8 is that there is benefit to applying the different operator merging optimizations in tandem. However, the incremental benefits are small as a law of diminishing returns sets in.

Whereas projections can be pushed into multi-way containment joins, set operations cannot be pushed in. In Table 5b we investigate this issue with respect to the union operator for different queries. Both Query 1 and 2 are three-node long chain queries. Query 1 has a lower nesting of the root. We find, as our analysis predicts, that the benefits due to pushing unions in could be more or less than the benefits due to using multi-way containment joins. The two optimizations are exclusive, and a good query optimizer will have to choose one or the other depending on its estimates of the cost for the particular data set and query.

6 Conclusion

XML query processing has been modelled in terms of macro-algebras (which operate on entire trees) and

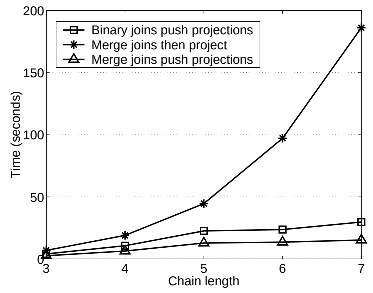


Figure 8: Comparing alternative optimizations as pattern chain length is varied

micro-algebras (which operate on individual nodes). In this paper, we have explored the relative ease of certain optimizations in one versus the other. 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, a negation containment join, and a multiway 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.

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

[3] 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.

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

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD*, June 1996.

[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] M. Consens and A. Mendelzon. Hy$^+$: A hygraph-based query and visualization system. In *Proc. SIGMOD*, pages 511–516, 1993.

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management systems for semistructured data. *SIGMOD Record 26(3)*, pages 54–66, 1997.

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

[17] 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.

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

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

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

[21] 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.

Appendix A: Stack-Based Negated Containment Join

In this appendix, we describe one of the containment joins presented in this paper, the *negated containment join*. We also discuss using it in pushing set difference into selections. The negated containment join reads in two lists, a list (AList) of potential ancestors and of potential descendants (DList). It outputs nodes from AList that are *not* ancestors of any node in DList using a stack data structure to keep track of ancestors. For example, consider a query that seeks employees who do not have a phone number. The algorithm would read in from an index all elements that matched the tag employee and from another index all elements that matched the tag phone-number. Afterwards, it would output employee elements that do not have a descendant phone-number.

Each node in the document is assigned a pair of numbers; a start key and an end key. These two numbers correspond to the offsets of the start and end tags of the element in the XML document. Therefore, an ancestor has a start key less than the start keys of all of its descendants and an end key that is greater than the end keys of all of its descendants. So, for a given pair of nodes, comparing the two start keys and the two end keys would decide whether or not one node is an ancestor of the other.

Fig. 9 presents pseudo-code of the stack-based negated containment join algorithm. Following up on the employee-phone example, AList would be the list of index entries that matched the tag employee. DList would be the list of index entries that matched the tag phone-number. The algorithm starts by reading in the first values from both lists. If the stack has elements in it, it checks if any of them is ready to be popped (lines 4-10 in Fig. 9). An element is to be popped if it has an end key smaller than the start key of either of the inputs. When it is time to pop an element from the stack, a flag (marked) associated with each stack element is checked; if it is still in its initial state (false), then this element did not join with any descendant and therefore should be output. The next step in the algorithm deals with either the potential ancestor or the potential descendant depending on which has the least start key (line 11). If the ancestor node (a) comes first, it gets pushed into the stack, its flag gets initialized to false, and the next ancestor node is read in. If the descendant node (d) comes first, we know that all stack elements are ancestors of d. Therefore, none of them should be output and their flags are set to true (line 19). Afterwards, the next descendant is read from the index. Even though this algorithm is somehow similar to the ancestor containment join presented in ??, it performs much better because it does not keep lists of output. It is closer to the partial join version of the ancestor containment join.

This algorithm is used to merge selections with set difference. Back to our example query; we want all employees who do not have phone numbers. This translates into a query plan that performs set difference between employees and employees with phone numbers. This is the upmost plan in Fig. 10. The selection is basically a sequence of binary containment joins. Therefore, the selection on the right side of the difference is broken into a single binary containment join with employee as ancestor and phone-number as descendant. This is the second plan in Fig. 10. The final step is to replace the difference followed by a containment join with the negated containment join, which is the bottom plan in Fig. 10. Comparing the second and third plans, we find that the third plan obviously wins because it has less number of containment joins and less number of index reads.

```
Algorithm Negated-Containment-Join (AList, DList) {
/* AList is the list of potential ancestors sorted by dist.start */
/* DList is the list of potential descendants sorted by dist.start */
/* Each entry in each list is a vector of elements, one of which,
called dist, is the one that participates in the join. */
/* We want to keep elements of AList that
DO NOT have descendants in DList*/
1.  a = Alist.first;
2.  d = Dlist.first; //cursors for the lists
3.  while (a & d) {
4.       while ((a.dist.start > stack.top.end) OR
5.               (d.dist.start > stack.top.end)) {
6.           //time to pop the top element in the stack
7.           if (stack.top.marked == false)
8.               output(stack.top)
9.           stack.pop();
10.      }
11.      if (a.dist.start < d.dist.start) { // process ancestor node
12.          stack.push(a.dist);
13.          stack.top.marked = false //initialize marked as false
14.          a = AList.next;
15.      }
16.      else {// process descendant node
17.          for each element i on stack if (d joins with a.dist)
18.                  // mark the element as NOT ouput
19.              stack.element(i).marked = true
20.          d = DList.next ;
21.      }
22. }
}
```
Figure 9: Negated Containment Join Algorithm

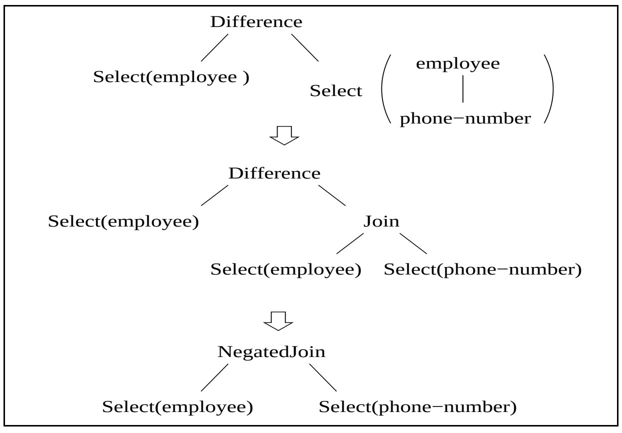


Figure 10: Steps in pushing difference in selection

Appendix B: Derivation of Algorithms Cost Formulae

In appendix B, we present the derivation of formulae in Table 1. An initial discussion of the formulae is in section 4.4.

Binary Containment Joins

The basic full binary containment join, with output sorted by descendant, requires reading the two input lists of candidate nodes and then writing out the result. There is no memory storage except for the small stack. If the ancestor candidate nodes have previously been joined multiple times, we could have a large value for v_1, which makes each stack entry very large. Even so, we do not expect the total size of stack to occupy a significant amount of buffer. The input cost of this join is $v_1 * n_1 + v_2 * n_2$ where n_1, n_2 are the cardinalities of the two inputs respectively, and v_1, v_2 are their respective vector sizes in number of elements. The output cost is $(v_1+v_2)*n_{12}$ where n_{12} is the cardinality of the output.

Considering the full binary containment join, with output sorted by the ancestor, there could be the need to write out intermediate results. In the worst case, this could be as large as the final output written out and read back once. In the best case, this is zero. The best case is actually not infrequent – it occurs whenever we can be guaranteed that no two instances of the sort key in the candidate ancestor set have an ancestor-descendant relationship. This happens because the bottom element in the stack does not retain lists, it outputs results immediately. We model this by introducing a nesting factor, f_1, that takes a value between 0 and 1. The cost of the intermediate write is $(v_1 + v_2) * n_{12} * f_1$, and the cost of the intermediate read back is also the same. Thus the total cost is $v_1 * n_1 + v_2 * n_2 + (v_1 + v_2) * n_{12} * (1 + 2f_1)$.

In a partial binary containment join, the vector size of the output is decreased to $u \leq v_1 + v_2$. The cardinality of the output is also decreased, to $m \leq n_{12}$. The input costs remain unaffected. The total cost is $v_1 * n_1 + v_2 * n_2 + u * m * (1 + 2f)$, with f_1 forced to zero if the output is sorted by the descendant node in the join. Comparing this with the formula for the full binary containment join, we see that the savings due to the partial join are greater the smaller the value of m, of u, and the larger the size of the output compared to the input. The savings are also greater when the output is sorted by ancestor rather than descendant, and greater when the sort key is more deeply nested.

The formula for negated binary containment join computation is exactly the same as for the partial binary containment join, except that u and m have correspondingly different meanings. Also, $v_2 = 1$ because there is no point in keeping nodes of the negated part except the one that is actually joining with the ancestor. Furthermore, the output vector is v_1 only. All the trends described above for partial binary containment joins apply to negated joins as well.

Multiple Containment Joins

First, we will discuss the multi-way chain containment join. Consider a chain pattern with three nodes and two edges. Each of the three nodes has sets of candidate matches that must be input, at a cost of $n_1 + n_2 + n_3$. (The vector size is just one in each case, if this is the entire pattern. Otherwise, it is v_1, v_2 and v_3, respectively). The output size is $(v_1 + v_2 + v_3) * n_{123}$, where n_{123} is the output cardinality. Again, f_1 is the nesting factor of the top ("ancestor-most") node in the chain. The disk I/O involved is simply $(v_1 + v_2 + v_3) * n_{123} * (1 + 2f_1)$. This is always a savings over performing the chain using two binary joins (where disk I/O may occur while joining the middle node with the leaf node as well as while joining the root with the middle node). Therefore, the formula for the three-way chain containment join is:
$v_1*n_1+v_2*n_2+v_3*n_3+(v_1+v_2+v_3)*n_{123}*(1+2f_1)$

A three node twig pattern has two descendants sharing the same ancestor. A twig join can be performed as one three way twig join. The input cost is $n_1+n_2+n_3$. Output cost is again $(v_1+v_2+v_3)*n_{123}$. The output cost is $(v_1 + v_2 + v_3) * n_{123} * (1 + 2f_1)$. Unlike the previous algorithms though, the bottom-most element in the stack needs to retain lists of inputs to make sure that all matches are found. We introduce another factor, k_i $(i = 2, 3)$. k_i (between 0 and 1) is the fraction of input i that participates in the output. This includes parts of the input that are considered for output even though they did not actually participate in it. For example, a descendant from input i comes and the stack is empty. This descendant is *not* part of $k_i * n_i$. On the other hand, a descendant that comes and the stack size is greater than one is always part of $k_i * n_i$ even if it does not become part of the output. For bookkeeping of input lists, the amount of I/O needed is $k_i * v_i * n_i$ for each input i $(i = 2, 3)$. Therefore, the formula for the twig three-way containment join is:
$v_1 * n_1 + v_2 * n_2 + v_3 * n_3 + (v_1 + v_2 + v_3) * n_{123} * (1 + 2f_1) + k_2 * v_2 * n_2 + k_2 * v_2 * n_2$

Pushing in projections will simply reduce both $v_1 + v_2 + v_3$ to u and n_{123} to m in the multi-way containment joins.

The stack-based containment join access method is equally applicable to native XML databases and to relational implementations of XML data management. As such, the analysis presented in this section apply equally to a variety of XML implementations.