Putting XML Query Algebras into Context

Stratis D. Viglas Leonidas Galanis
David J. DeWitt Jeffrey F. Naughton
University of Wisconsin—Madison
Department of Computer Sciences
e-mail:{stratis, lgalanis, dewitt, naughton}@cs.wisc.edu

David Maier
Oregon Graduate Institute
Department of Computer Science
e-mail: maier@cs.ogi.edu

Abstract

While the XML community appears to be converging on XQuery as a standard for querying XML documents, there is currently much less consensus about a standard algebra. In this paper, we describe the algebra we have implemented in our XML query evaluation system. Our goal was to specify an algebra that is powerful enough to handle the XQuery language, yet simple enough to be amenable to optimization and implementation. A novel aspect of our algebra is the use of the context construct. Whereas XML algebras using bindings must at some point resort to "shredding" XML into collections of tuples that may need to be recombined by subsequent operators (often, for example, in a result construction step), the context construct allows us to "carry along" the structure of the input XML in a way that is transparent to the optimizer and to later operators in the query plan. It is our hope that publishing our algebra, and the context construct in particular, will contribute to the discussion of what features and characteristics an XML algebra should possess.

1 Introduction

Over the past couple of years, the database community has focused much attention on mechanisms for querying XML. The result has been a number of XML query languages, and a number of prototype implementations of these query languages. However, although there is growing consensus on XQuery as a standard query language, there is to date no accepted "XML query algebra." This situation is in marked contrast to relational systems, in which relational algebra has long been recognized as a useful intermediate form in the process of query execution. In this paper we describe our candidate XML query algebra, with the hope that this paper can contribute to the discussion of what features and characteristics an XML algebra should possess. We give a precise formal description of our algebra, and outline an algorithm that translates from XQuery to our algebra.

Our goal was to specify an algebra that is powerful enough to handle the XQuery language, yet simple enough to be amenable to optimization and implementation. Moreover, it should be capable of exploring the structure of the data, as well as querying on their values (as per the desiderata in [Mai]) since the majority of XML data we have come across, although implicitly adhering to a schema, do not explicitly specify that. A novel aspect of our algebra is the structure we use for intermediate results, which uses a tree-structured context instead of a fixed vector of bindings. XML queries make heavy use of paths, and algebras tend to support that with unnesting and path-extraction operators to get at sub-elements.

Representing the nodes so accessed using a fixed vector of bindings means a possible loss of nesting and grouping relationships, and extra work in order to recover them. Using a context, as we shall see, allows such connections between accessed nodes to be preserved.

The rest of the paper is structured as follows. Section 2 further motivates the need for a context construct in XML query algebras. Section 3 presents a formal XML data model, while the formal definition of context and the algebraic operators is defined in Section 4. Section 5 presents some of our experiences with implementing the proposed algebra in the context of our XML query engine, while Section 6 presents related work. Finally, Section 7 summarizes the framework and identifies our planned future directions. The Appendix provides a translation algorithm from XQuery to our algebra.

2 On the Need for Context

A novel and key aspect of our algebra is the use of a construct we term a *context* as a way to specify, access and manipulate different XML elements. Contexts replace the more commonly used bindings. Algebras with bindings use variables to represent state in the query tree. At run time, these variables will be bound to sets of XML fragments; at query optimization time, the "meaning" of a variable can only be extracted by finding and analyzing the point lower in the tree containing an assignment to that variable. In contrast, our algebra uses contexts to represent state. We give a formal and detailed definition of contexts in Section 4, but, informally, a context represents a point in an XML document during query evaluation. The context at an operator encapsulates the path taken through the source documents to reach the inputs to this operator.

We think the use of contexts instead of bindings is beneficial during both query optimization and query execution. We suspect contexts will be useful at query optimization time because they make state transparent and local to an operator in the algebraic query tree. This transparency is likely to lead to a simpler and more powerful optimizer than one that has to decode a chain of variable bindings to analyze the state at each operator. To see the benefit of contexts at query evaluation time, we turn to the following example.

```
<bibliography>
      …
      <proceedings> // 10 occurences
            …
            <editor> … </> // 3 per proceedings
            <editor> … </>
            …
            <article> // 4 per proceedings
                  <author> … </> // 2 per article
                  <author> … </>
                  …
            </article>
            …
      </proceedings>
      …
</bibliography>
```

Figure 1: A sample bibliographic database; elements are annotated with their cardinality

Consider the XML document fragment in Figure 1, in which we have a collection of `proceedings` elements and their `editors`, as well as the list of `articles` published in those `proceedings` along with the list of `authors` for each `article` (the elements in Figure 1 are annotated with their cardinalities). Now, suppose we want to pose the query "*find all authors who have not had a publication in proceedings they have edited,*" which requires calculating the intersection between the `author` and `editor` collections of each `proceedings` element. Let us now assume that we were trying to express the query in a relational-style algebra with bindings, augmented with *unnest* operations bearing the following semantics: given a containing element x and a path expression y, *unnest*(x, y) generates pairs of (x, y) elements for each contained element e reachable by path expression y.

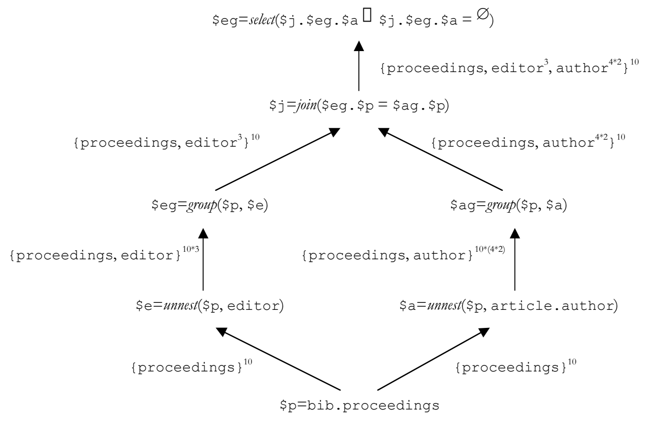
Figure 2 shows how our query would be represented in such an algebra, using the notation that `$x` denotes a binding to variable `x`, `tag`^{y} corresponds to y occurrences of element `tag`, and in $\{\}^z$, z is the cardinality of the given set. In this kind of algebra, we first create a binding for all `proceedings` elements and then reach into each element, unnesting the `editor` and `author` sub-elements. This creates two separate sets of `editors` and `authors`. The next step is to group each set by their `proceedings` elements, and join them to create a set of `proceedings` elements with all their `editors` and `articles`' `authors` elements at the same level. Finally, we select the sets of groups for which the subset relationship holds. In total, this tree uses seven operations and six intermediate bindings. Also note that in formulating the expression we had to undo an operation - when we unnested `author` and `editor` elements, we lost track of the `proceedings` element in which they appeared. We had to re-create the nesting through the subsequent grouping and join operations. Our goal will be to use context to avoid these problems.

Now, consider the algebraic tree of Figure 3, which uses an informal algebra that exploits context instead of using bindings. Once again we begin by accessing the `proceedings` elements. We can use the proceedings elements as a context within which to access `editor` sub-elements, creating a wide set consisting of a `proceedings` element and all its `editor` sub-elements. The set maintains the association between the editor sub-elements and the `proceedings` element, so we can reuse it as a context, extending the set with all the `author` sub-elements. We now have an even wider set, consisting of a `proceedings` element, and two subsets: one for `editors` and one for `authors`. The only remaining operation is an intersection and selection over the wide set.

In the approach with context, we have used four operations and no bindings for temporary results. Two observations allowed us to do so: Whenever we have to access multiple sub-elements of the same element, we should not lose track of the nested structure. Moreover, as long as we have a way to uniquely identify the set element we operate on, we do not need bindings. Our notion of *context* encapsulates both of these observations. In the aforementioned example, the `proceedings` element is the context of the extension operations, while the contexts for the intersection operation are the `author` and `editor` sub-elements.

Although the focus of this paper is not on evaluation algorithms, we would like to touch on at least one issue related to efficiency. At first glance, it may appear that the use of context is inefficient, since large amounts of state need to be passed between the operators of an evaluation tree. For example, after an unnest operator, the parent of the unnested nodes is apparently replicated as context for each of the children. This depiction is misleading – in actuality, the state is not replicated, instead, the children share the context, so the context is expanded at the cost of only one pointer per child. While contexts are used uniformly at the logical level of the algebra, there are alternative means of representing them in physical plans. Some of these alternatives, when they apply, are comparable in efficiency to bindings.

We close this section by stating the obvious – we are not claiming that an algebra with context is more powerful in a computability theoretic sense than an algebra with bindings. That is certainly false, since if two algebras can both express the same set of queries, presumably one could write a translator that maps from a query in one algebra to an equivalent query in the other. Rather, we are arguing that the use of context makes expressions simpler and more transparent, and that designing and implementing powerful optimizers and efficient query evaluators will be easier with the use of context than with the use of bindings.



$eg=select($j.$eg.$a ⊆ $j.$eg.$a = ∅)
{proceedings, editor³, author⁴*²}¹⁰
$j=join($eg.$p = $ag.$p)
{proceedings, editor³}¹⁰
{proceedings, author⁴*²}¹⁰
$eg=group($p, $e)
$ag=group($p, $a)
{proceedings, editor}¹⁰*³
{proceedings, author}¹⁰*(⁴*²)
$e=unnest($p, editor)
$a=unnest($p, article.author)
{proceedings}¹⁰
{proceedings}¹⁰
$p=bib.proceedings

Figure 2: A relational-style algebra's computation tree

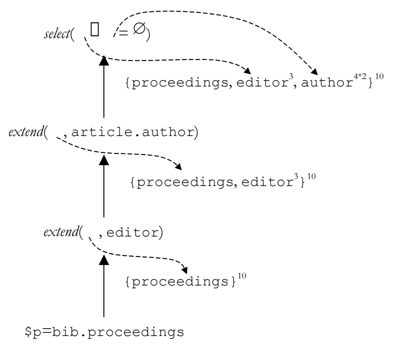


Figure 3: A context-aware algebra's computation tree. Dashed lines represent context specification.

3 Preliminaries – a Data Model for XML

Before describing our algebra, we first review a precise definition of the XML data on which our algebra operates. This definition is not intended to be a contribution of this paper – any precise way of describing XML data would do. Rather, the importance of describing this model is that it lets us present our algebra in a formal and unambiguous way.

3.1 The XML Data Model

Every XML document adheres to a data model defined in the XML standard specification [BPM98]. The model a document by a directed rooted graph. Figure 4 shows an example XML document, named `bibliography.xml`, with a topmost, unique *element*, `bib`, known as the root of the document. This root encloses all sub-elements. Three `book` sub-elements reside within the root; in general, nesting can go on to an arbitrary level. It is possible to have *attributes* associated with elements; for example, the `book` element has the attribute `isbn`.

Figure 5 depicts the graph model for the document of Figure 4. Elements and attributes correspond to vertices in the graph. Directed, labeled edges connect the vertices, with the tag of the corresponding element or attribute name acting as the label of the edge. For each vertex of the graph except the root, there is a backward edge leading to the parent vertex. A third type of edge, not depicted in the example, is a reference edge, which can point to an arbitrary element. Note that parent edges appear only between vertices that are connected with element or attribute edges. If we assume an ordering on sibling edges in the graph, we can define a total ordering on vertices by a depth-first traversal of the graph.

A path expression consists of the sequence of edge names one needs to follow in order to arrive at a vertex. This path expression is not guaranteed to be unique, since the same vertex might be reachable by more than one path expression (for instance, if the vertex is the ending point of multiple reference edges).

```
<bib>
  <book isbn="0-201-53771-0">
    <title>Foundations of Database Systems</title>
    <author>
      <lastname>Abiteboul</lastname>
      <firstname>Serge</firstname>
    </author>
    <author>
      <lastname>Hull</lastname>
      <firstname>Richard</firstname>
    </author>
    <author>
      <lastname>Vianu</lastname>
      <firstname>Victor</firstname>
    </author>
  </book>
</bib>
```

Figure 4: An XML Document fragment

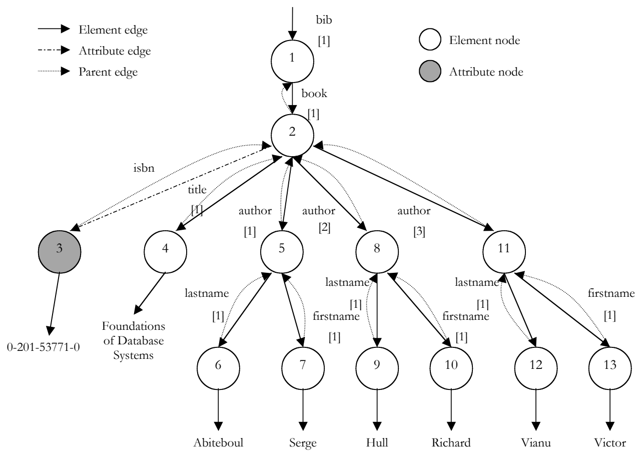


Figure 5: The data model corresponding to Figure 4's document fragment

3.2 Type Hierarchy

Our algebra makes use of a simple type hierarchy for XML. For our purposes, we do not need to distinguish between different types at the specific XML element tag level. That is, we consider the `book` and `author` elements in Figure 5's example to be of the same type, regarding them simply as *elements*. There are two distinct types on which our algebra operates, the *Text* and *Node* types. *Text* corresponds to the textual representation of XML data. For instance, "Foundations of Database Systems", "Abiteboul", "Vianu" and "Hull" are all textual entities. We use the *Text* type for value comparisons between XML data. The other type we operate on is the *Node* type, which encapsulates all composite structures appearing in an XML document: elements, attributes and plain data. A *Node* can be refined in one of two ways: it is either a *Tagged* node, meaning the edge that leads to it is named; or it is *Untagged*, in which case the connecting edge is unnamed. The second possible refinement is between *Flat* and *Composite* structures. *Flat* structures provide for no nesting, whereas *Composite* structures can have nested structure. Figure 6 depicts the type hierarchy of our framework. Two

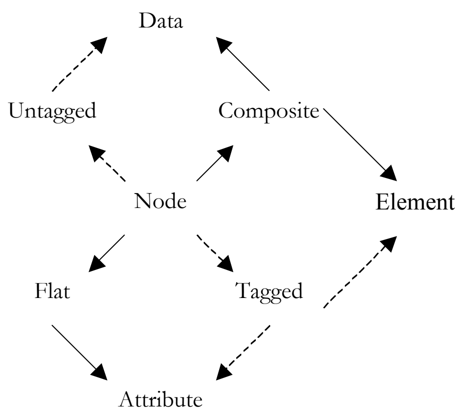


Figure 6: Type hierarchy

additional primitive types we use are the *Integer* and *Boolean* data types. The former is used for *order comparison* (see Sections 3.3 and 3.4) while the latter is used in path expression and predicate evaluation (see Section Collections). Given this type hierarchy, we can move on to introduce the type constructors, which are summarized in Table 1.

The *Data* type constructor accepts a *Text* parameter and generates a data node. The *Attribute* type constructor takes as input two *Text* elements and produces a new *Attribute* with the first *Text* parameter becoming the tag of the attribute and the second *Text* parameter its value. The *Element* type constructor takes as input a *Text* parameter for the element tag, a vector of *m* attributes for the element, and a vector of *n Composite* objects that become the content of the element.

Data type	*Type constructor*
Data	*Text* □ *Data*
Attribute	*Text* □ *Text* □ *Attribute*
Element	*Text* □ $Attribute^m$ □ $Composite^n$ □ *Element* (m, $n \geq 0$)

Table 1: Type constructors

3.3 Functions

To allow us to describe how our algebraic operators access their input, we define five functions that operate on XML nodes: *id()*, *tag()*, *value()*, *content()* and *parent()*. The *id* of a node is the value of the total ordering the node has in the XML graph[1]. For example, calling *id()* on the first `author` element of Figure 5 will return *5*. The *tag* of a node is the name of the edge that points to it in the XML graph. Calling *tag()* is only meaningful for *Tagged* nodes. The *value()* function normalizes a node into a textual representation that is used for value-based comparison. In Figure 5, calling *value()* on the first `author` element yields: `<author><lastname>Abiteboul</lastname><firstname>Serge</firstname></author>`. The *content* of a node is everything that appears below the node in the XML graph. For instance, for the graph in Figure 5, calling *content()* on the first `author` element will yield a set consisting of the `lastname` and `firstname` elements corresponding to `Abiteboul` and `Serge`. The last function is *parent()*, which returns the parent node of any given node. In Figure 5, calling *parent()* on the first `author` element returns the containing `book` element.

3.4 Predicates

The operators in our algebra make use of two types of predicates: *Boolean* predicates and *text-in-context* predicates. For the former, we support all value comparison predicates (=, >, <, □, ≥, ≠) between the results of any of the supported functions from Section 3.3. For instance, to support comparison on *id*s we could use a predicate *id*($node_a$) = *id*($node_b$), which would succeed only if the participating nodes are the same *physical* node. The supported functions are also useful in handling features such as *tag queries*, in which case we test predicates on the tag of any given node. As an example, *tag*(node) = `author` would succeed only if the tag of the node was equal to `author`. Comparison on *id*s allow us to handle specific

[1] Which the data model guarantees to be unique.

query language constructs, such as XQuery's *before* and *after* predicates (see also the Appendix), while comparison on *tag*s supports some of the more obscure features of XML querying.

Finally, we support as primitive predicates some "text-in-context" predicates, i.e., *contains*, *contained in*, *directly contains*, *directly contained in* and *distance to.* For instance, we can have predicates of the form *node contains* `author` which succeeds only if the left-hand *node* contains under any level and under any type (meaning, as an element, as an attribute or as plain data) the text `author`. These text-in-context predicates are defined to allow our algebra to more naturally express certain types of "structured text document" queries.

```
result := {};
for all x in A do
        result := result $\cup$ f(x);
end for
return result;
```

Algorithm 1: A mapping homomorphism from lists to sets

3.5 Collections

Our algebra uses three primitive collections: *bags*, *sets* and *lists*. The input and output to all operators are *collections of collections*, in most cases, *sets of sets*. For brevity, in what follows we will use the terms *outer collection* or simply *outer* for the enclosing collection and *inner collection* or simply *inner* for enclosed collections. Essentially, each outer collection consists of structures like those in Figure 7, with each such structure modeled as an inner collection. Whenever we have to switch between collection primitives, we employ an appropriate *homomorphism* that maps one collection primitive to another, in the style of [FM95a] and [FM95b]. For instance, we can map from a list to a set with a homomorphism of the form $hom^{list \rightarrow set}(f)A$ where A is the list and f is the mapping function, as shown in Algorithm 1. In what follows, we will use $\{\!\{\}\!\}$ to signify a bag, {} to denote a set, while [] symbolizes a list. Homomorphisms allow us to do without a final frequently used type of collection, the *sequence set*, which can be captured as the homomorphism $hom^{bag \rightarrow set}(hom^{set \rightarrow list})(f)A$.

4 Context and Operators

In this section we describe the main features of our algebra: the context construct and the operators.

4.1 Context and Path Expressions

The main job of the context construct is to specify a position within an XML graph. In addition to designating nodes in the graph, this construct gives context to those nodes by describing how the query evaluation arrived at the nodes. We represent this context with *regular path expressions.* Before turning to path expressions and how they are used, we digress to describe how contexts are used to describe the input to operators.

The basic idea is that each input to an operator is described by its context, but that this is not sufficient – we also need to a mechanism to describe how the operator should "explore forward" from this context as it performs its specified function. We also use path expressions to describe this forward exploration, and combine it with the context. One way to view this combined object is as a single regular path expression with two distinct parts: a context path and a forward path. The notation we use is [*context*]*forward*, which roughly translates to all elements reachable by the forward path, whose relative location in the graph is specified by the context. We cannot simply use the full path from the root as our node specification mechanism, since this would disregard any intermediate results we have already generated through prior navigation.

For instance, suppose we want to reach the `lastname` element in the XML document of Figure 5. Through previous navigational steps, we have generated intermediate results (hence, new context) on the root element of the file, the `book` and `author` sub-elements. The question then becomes how to specify the context we should use to reach the `lastname` element. One way to do so is to use the expression `[□]<bibliography.xml>.book.author.lastname`[2], which uses the *root* element of the document as context. A second way is `[<bibliography.xml>.book]author.lastname`, which uses the `book` element for context, while a third is `[<bibliography.xml>.book.author]lastname`, using as context the `author` element.

Figure 7 depicts these path expression examples in our path expression framework; a square represents a *context* node, while an oval represents a *navigation* node. The operators in our algebra manipulate structures like the one in Figure 7, both by extending them through creating new context and by navigating to new parts of the XML document. For generality, we assume all XML nodes appear below a universal *empty context* node (□). Notice that whenever we navigate to a new node, that node shares context with its ancestors. Whenever we wish to evaluate a path expression, we are in essence evaluating both the context *and* the relative navigation part. In what follows, we will be using pictures like the one in Figure 7 to describe our operators. Our inners collections, however, provide direct access to the XML nodes immediately below context nodes. While we draw an inner collection as a tree for understandability, it is really a collection of (context, navigation) pairs.

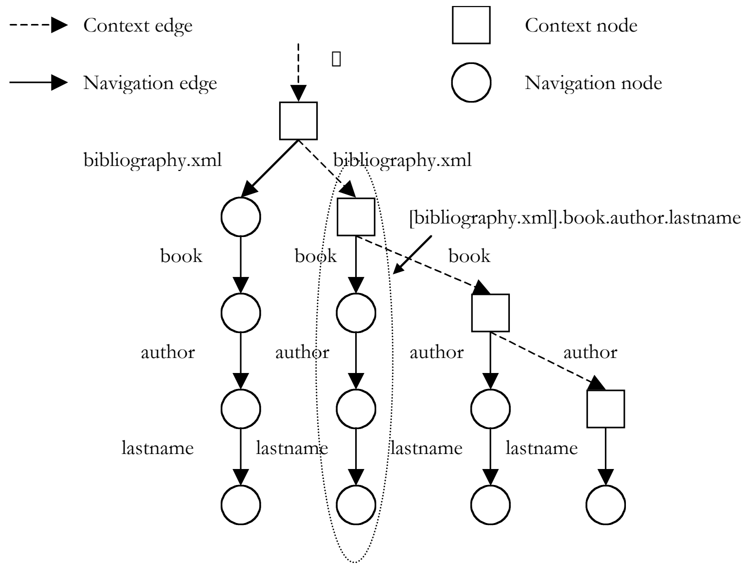


Figure 7: The pictorial representation of path expressions

[2] The symbol □ denotes the empty path expression.

4.1.1 Path expression evaluation

In this section we give further information on the semantics of our path expressions by describing one alternative for their evaluation at runtime. For brevity, we focus on the special case of regular path expressions involving Kleene-star. The input to the path expression evaluator is an XML node and a path expression combining both the context and the forward part. We will use small characters from the beginning of the alphabet such as *a* and *b* to represent bound constants (in most cases, specific tags appearing in the document). Small letters near the end of the alphabet such as *x*, *y* and *z* denote graph nodes, while capital letters such as *A* and *B* denote complete path expressions. We break the evaluation into two phases, one that evaluates the context part (*cpe()*) and one that evaluates the relative forward part (*fpe()*). The context evaluator returns a *boolean* value stating whether the initial node satisfies the context part and, if that succeeds, the forward path expression evaluator returns the set of all nodes that are reachable. Algorithm 2 provides a sketch of the evaluation predicate, *evalPE*. In the predicate, *x* is an XML node, while [*B*]*A* denotes the context to be evaluated.

```
evalPE(x, [B]A) □  {y | cpe(x, B) □ y □ fpe(x, A)}.
cpe(x, □) □  true.
cpe(x, b) □  (tag(x) = b).
cpe(x, B.b) □  (y = parent(x)) □ (tag(y) = b) □ cpe(y, B).
cpe(x, B.*) □  (y = parent(x)) □ (cpe(y, B)    cpe(y, B.*)).
fpe(x, □) □  {x}.
fpe(x, a) □  {y | y □ content(x) □ (tag(y) = a)}.
fpe(x, a.A) □  {y | z □ content(x) □ (tag(z) = a) □ y □ fpe(z, A)}.
fpe(x, *.A) □  {y | z □ content(x) □ y □ {fpe(z, *.A) □  fpe(z, A)}}.
```

Algorithm 2: Path expression evaluation

One can generalize the algorithm to provide support for other features, such as having regular expressions as tag placeholders in the path expressions. Due to space limitations, in this section we try to give a feel for how evaluation takes place rather than describing our full path expression evaluation algorithm.

Notation	*Description*	*Example*	*Explanation*
[n]	The *n*-th child of a node	`[bib]book.author[2].lastname`	The `lastname` of the second `author` of a `book`
#	The whose name follows must lead to an element edge	`[bib]book.#author`	The `author` sub-element of a `book` element
@	The edge whose name follows must lead to an attribute node	`[bib]book.@isbn`	The `isbn` attribute of a `book` element

Table 2: Syntactic constructs to restrict the nodes specified by a path expression

We can also extend our path expressions to use the syntactic constructs of Table 2. Although these constructs are not the preferred way of node specification, as they compromise the exploratory nature of our algebra, we have decided to add them to provide closer support to specific features found in query languages make use of the XPath standard (including XQuery.) We evaluate these path expressions by calls to the functions of Section 3.3 augmented with node type checking (i.e. whether they are attributes, elements or data nodes).

Finally, we need to describe how are contexts interact with the collections that are passed about in the query evaluation tree. When defining operators, we treat each inner collection as a collection of (*context*, *navigation*) pairs. Inner collection participation is then based on *node id* of the navigation part's final destination. For multi-level collections, participation is defined recursively: outer collections are equal if they contain equal inner level collections, with inner collection equality determined as equality over all participating elements.

4.2 The Operators

In this section we define the operators of our algebra. All operators accept a collection as input and produce a collection as output. Our operators concentrate on pattern retrieval, filtering and construction. In what follows, we assume that the initial input to an algebraic expression is a set containing singleton sets whose unique element is the document element of the documents we operate on. For example, an expression operating on `bibliography.xml` would have as initial input a set containing a singleton set containing a pointer to the root element of document `bibliography.xml`. An expression operating on a set of XML files would have as initial input a set containing as elements a singleton for each XML files, each singleton containing the root element of one such file. We present all operators in the form *op*(*arguments*) *collection*, which means that the operation *op* with the given *arguments* is to be carried out over *collection*. Table 3 presents a summary of the notation we will use for the formal operator definitions.

4.2.1 Alias

The *alias* (symbolized as □) operator provides renaming at the tag leve. Its formal definition is:

$$□(old, new)C = \{K \mid K □ C, x □ K, (x/y □ \text{tag}(y) = old □ \text{rename}(y, new))\}$$

Aliasing is mainly used in self-joins to avoid name conflicts.

Symbol	*Description*
$x, y, z, \ldots$	XML nodes (regardless of type)
$a, b, c, \ldots$	Context XML nodes
$A, B, C, \ldots$	Outer collections
$K, L, M, \ldots$	Inner collections
$a □ A$	A ranges over A
P	A regular path expression
y/x	Node x is a child of node y
$p(x)$	A predicate that evaluates to true or false when tested on x

Table 3: Notation summary for operator definitions

4.2.2 Unnest

The *unnest* operator (symbol: □) expands the collections appearing in the input collection by the nodes reachable by the argument path expression. In case multiple nodes are reachable by the same path expression (as is the case in the document of Figure 5, which has three elements reachable by the path expression `[<bibliography.xml>.bib.book.author]`) then multiple new inner collections are created. The formal definition of the unnest operator is:

$$□(P)C = \{L \mid K □ C, (c, y) □ K, (x □ \text{evalPE}(y, P) □ L = K □ \{(y, x)\}\}$$

Notice that the unnest operator builds new context, as represented pictorially in Figure 8.

4.2.3 Select

The *select* operator (denoted by □) qualifies an inner collection if there exists some element in it that satisfies a specified predicate p. The formal definition is:

$$□(p)C = \{K \mid K □ \ C, (c, y) □ \ K, (p(y))\}$$

4.2.4 Project

The *project* operator (symbolized as □) strips away from an inner collection all irrelevant nodes, based on a list of path expressions that are passed as arguments. As is the case with *unnest*, *project* builds new context. The operator's formal definition is:

$$□(P_1, P_2, P_3,\ldots)C = \{L \mid L □ \ \{(y, x) \mid K □ \ C, (c, y) □ \ K, (x □ \text{evalPE}(y, P_i))\}\}$$

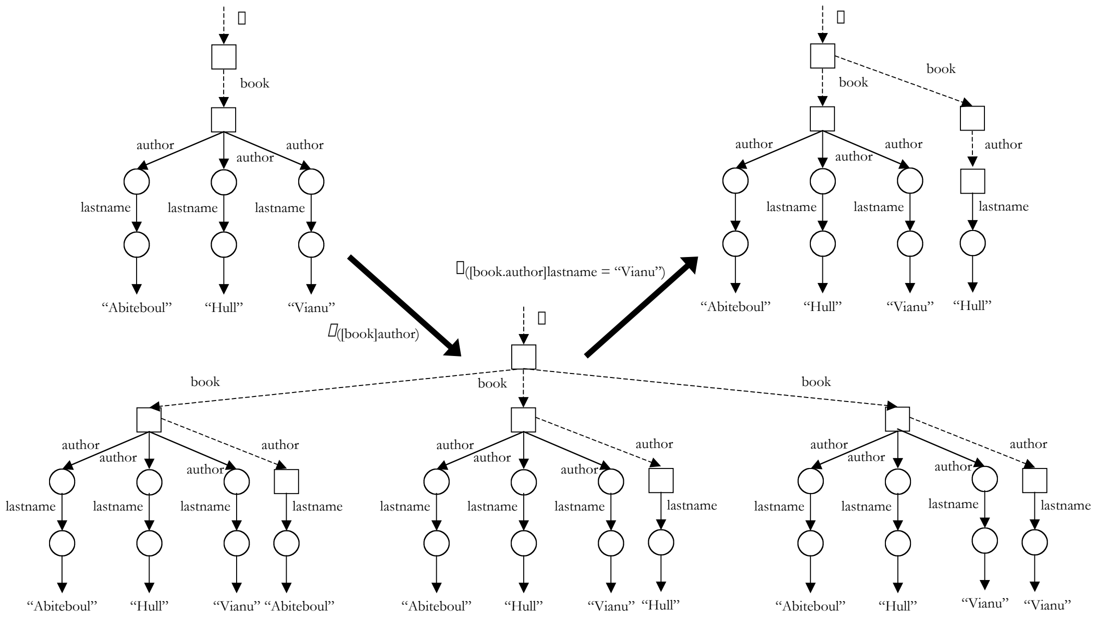


Figure 8: An example of an *unnest* operation, followed by a *select*

4.2.5 Join

The *join* operator (symbol: _) joins the inner collections of two outer collections if the individual inner collections satisfy the join predicate:

$$\_(p)(C, D) = \{M \mid K □ \ C, L □ \ D, (c, x) □ \ K, (d, y) □ \ L, (p(x, y) □ M = K □ \ L)\}$$

4.2.6 Extend

The *extend* operator (denoted by □) is one of the most powerful operators of our algebra. It accepts as its argument a mapping function *f* and applies that function to each inner collection, appending the result to that collection. The mapping function can be arbitrarily complex – in fact it can be a whole algebraic expression of its own.

$$□(f)C = \{L \mid K □ C, y □ K, (L = f(y) □ K)\}$$

One can think of the *extend* operator as a generalization of *project*. The mapping function for *project* is *evalPE()* and the difference is that while *project* trims the inner collection, *extend* simply concatenates the result to it.

4.2.7 Structural Recursion

Structural recursion (symbolized as □) is the second powerful operator of our algebra and is the way we introduce fixed-point semantics. One can think of structural recursion as the way to continuously apply a specific algebraic expression until no more applications of the expression over its argument collection are possible. The way to achieve this effect is to recurse on the elements of the incoming collections *a la Prolog*, as shown in the operator's formal definition:

$$□(E)C □ C = [K \mid A], □(E)\{K\} □ □(E)A$$

In the definition, E signifies an algebraic expression that is to be applied continuously over the incoming outer collection C, which we treat as a list. The latter is composed of an inner collection K, acting as the *head* of the list, and an outer collection A, acting as the *tail* of the list. We apply expression E to the singleton outer collection $\{K\}$ and to the tail of the incoming collection. The semantics of the operator are such that recursion terminates as long as E at some point produces an empty collection[3] (for instance, in the case of a selection predicate that evaluates to *false*.) Structural recursion appears necessary if an algebra is to be capable of supporting operations such as XQuery's *filter*.

4.2.8 Group

A *group* operation groups a number of inner collections into a single collection based on the evaluation of the output of a function. For instance, assuming we had unnested the `book` element of Figure 5 into its `title` and `author` subelements, we could recreate the original element by grouping on the *value()* of the `title` element, as shown in Figure 9. To formally define the operator we will use an auxiliary operator named *partition* – *p*, which has the following definition:

$$p(f)\ C = \{K \mid L □ C, M □ C, (a, x) □ L, (b, y) □ M, f(x) = f(y), K = L □ M - \{(b, y)\}\}$$

One can think of the partition operator as a single-level grouping operation. To extend that to the grouping operation in Figure 9, we need to use structural recursion.

[3] Recursion termination is similar to identifying whether a least fixed-point operation in relational algebra terminates. In that context, [T55] and [AU79] assure that if expression E is monotone in the context of a partial order _ between collections, a unique fixed-point exists.

$$\square(f)C = \square(p(f))C$$

A possible extension to the grouping operator provides room for aggregation operations. For instance, assuming each `book` element had a `price` sub-element as well and we had unnested `author` and `price` elements, we could calculate the average price of each author's books by a composite grouping operation of the form:

□([`<bibliography.xml>.bib.book.author`]□, *avg*([`<bibliography.xml>.bib.book.price>`]□))…

To fully specify aggregation in this way the definition of the group operator has to be extended to account for external functions being passed to it as arguments, which is outside of the scope of this paper.

4.2.9 Set Operations

In addition to the XML specific operators defined in the previous sections, we need the standard set theoretic operators to make the algebra complete. All set theoretic operators are based on the *id* of the participating nodes and not on their value. The operations we support are *union*, *intersection*, *difference* and *Cartesian product.*

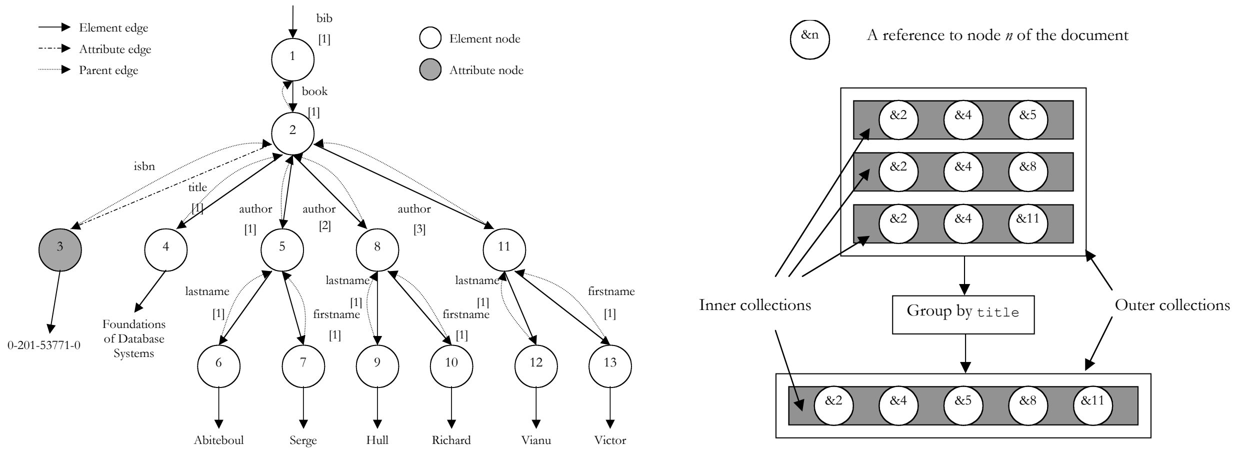


Figure 9: Group `book` elements by their `title` element. To simplify the figure, instead of presenting the actual elements we show them as references to the document's nodes.

4.2.10 Construction Operations

We incorporate three construction operators into our framework, one for each type of node we support. Each operator calls the appropriate type constructor (see also Section 3.2) and appends the constructed node to the incoming inner collection. Once the nodes are appended to the collection, they are accessible through a new path expression having an empty context. Table 4 shows the three construction operators. There are c_d for data, c_a for attributes, and c_e for elements, each one calling the appropriate node constructor, *data*, *attribute* or *element* respectively. Notice in Table 4 that all construction operators assign a default empty context (□) to the nodes they construct.

The data node construction operator, c_d, constructs a new data node by calling the appropriate type constructor for the specified text. The attribute construction operator c_a, accepts as its argument the tag of the attribute node to be constructed and a path expression that specifies the incoming inner collection's element that is to be used as the content of the attribute node. For instance, c_a(`my_isbn`, *value*(`[□]bib.book.isbn`)) constructs a new attribute node with a tag `my_isbn` and as its content, the value of the node specified by the path expression `[□]bib.book.isbn`. The constructed attribute is then accessible by the path `[□]my_isbn` (since new nodes are assigned an empty context). The element construction operator c_e, acts in the same way as the attribute construction operator except that it accepts as its arguments a list of attributes and a list of elements that will act as its content. The appearance order in the lists determines the order the nodes will have in the constructed element. For instance, calling c_e(`my_book`, [`[□]my_isbn`], [`[bib.book]title`, `[bib.book]author`]) will create a new element named `my_book` having as content an attribute node specified by `[□]my_isbn` and two sub-elements specified by `[bib.book]title` and `[bib.book]author`. Figure 10 shows the output of a construction operator sequence. Note that the textual parameter passed to each construction operator (i.e., the *text* for c_d, or the *tag* for c_a and c_e) does not have to be static text, it can be the result of any function called on a node, as long as it evaluates to *Text*. This allows us to capture arbitrarily complex construction operations, such as using the value of a retrieved element as a tag for element construction. For example, in the XML document of Figure 5, we can have an element construction operator that creates elements tagged with the *value* of a `lastname` element, i.e., elements tagged as `Abiteboul`, `Hull` or `Vianu`.

Operator	*Formal Definition*
$c_d(text)\ C$	$\{K \mid L □\ C\ (K = L □\ \{data(text)\})\}$
$c_a(tag, P)\ C$	$\{K \mid L □\ C,\ (a, y) □\ L,\ x □\ \{z \mid z □\ evalPE(y, P)\},$ $K = L □\ \{(□, attribute(tag, value(x)))\}\}$
$c_e(tag, [P_i], [P_j])\ C$	$\{K \mid L □\ C,\ (a, y) □\ L,\ X_a = [(□, z) \mid z □\ evalPE(y, P_i)],$ $X_e = [(□, z) \mid z □\ evalPE(y, P_j)],$ $K = L □\ \{(□, element(tag, X_a, X_e))\}\}$

Table 4: Formal definition of constrution operators

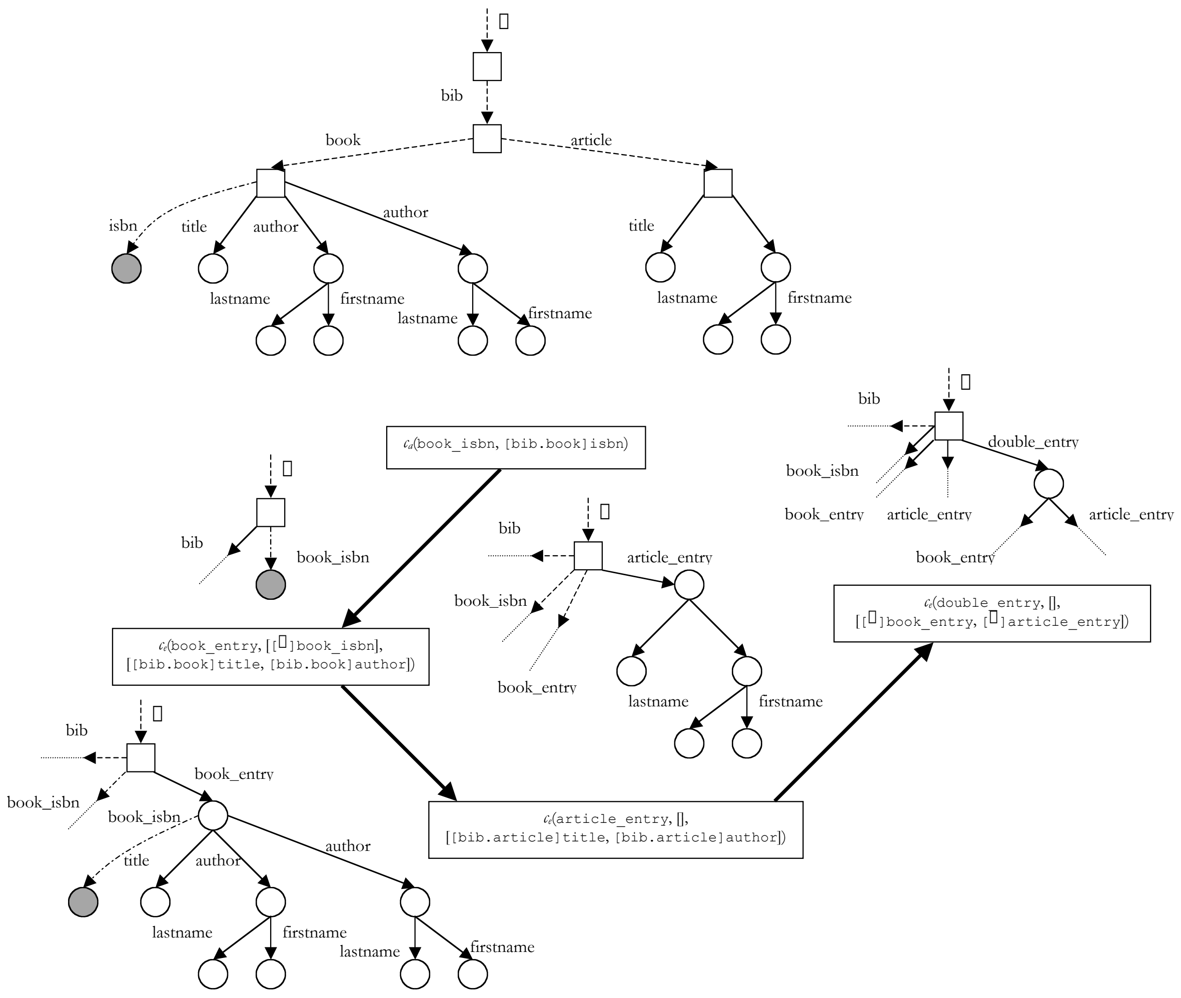


Figure 10: An elaborate construction operation. We present an initial inner collection as an input and see how it is modified over operation execution. At each step, only the constructed nodes are presented. For simplicity we have omitted the data nodes and the parent edges.

4.3 Examples

In this section we present two indicative examples of how we can translate XQuery queries into algebraic expressions.

The document of Figure 4 groups authors by book titles. Suppose we want to change the grouping to titles by book authors. Figure 11 presents a query that provides the way to do so through an XQuery query, with the corresponding algebraic expression.

The proposed algebra can also capture other, more advanced XQuery queries. Figure 12 presents the query: "*Find item numbers and descriptions of Bicycles offered by users whose rating is 'A', and the names of the offering users*" and its algebraic equivalent.

5 Opportunities for Optimization

We have implemented a large subset of this algebra in our XML query processor, and use this subset as the target language in which logical plans are expressed. In our implementation, the logical plans are then translated into execution plans by a straight one-to-one mapping, as our optimizer is still under construction. In this section we describe our initial thoughts on the opportunities for optimization presented by our algebra.

```
FOR $a IN
    document("books.xml")//book/author
RETURN
  <BooksByAuthor>
    <Author> $a/text() </Author>
    (
      FOR $b IN
          document("books.xml")//book[author=$a]
      RETURN $b/title
    )
  </BooksByAuthor>
```

```
□([BooksByAuthor]□) {
        ce(BooksByAuthor, [], [[author]□, [title]□]){
        ce(author, [], [[<books.xml>.*.book.author]□]){
        ce(title, [],[<books.xml>.*.book.title]□]){
        □([<books.xml>.*.book.author]□){
        □([<books.xml>.*.book]title){
        □([<books.xml>.*.book]author){
        □([□]<books.xml>.*.book){
        books.xml}}}}}}}}
```

Figure 11: An example XQuery query which transforms the element grouping from *books by title* to *books by author* and its corresponding algebraic expression

```
<result>
  (
   FOR $i IN
          document("items.xml")//item_tuple
       $u IN
          document("users.xml")//user_tuple

   WHERE $i/offered_by = $u/userid
   AND contains($i/description, "Bicycle")
   AND $u/rating = "A"

   RETURN
     <interesting_item>
       $i/itemno, $i/description
       <seller> $u/name/text() </seller>
     </interesting_item>
  )
</result>
```

```
□(interesting_item){
        ce(interesting_item, [], [
                [<items.xml>.*.item_tuple.itemno]□,
                [<items.xml>.*.item_tuple.description]□,
                [seller]□]){
        ce(seller, [], [<users.xml>.*.user_tuple]){
        (□[<items.xml>.*.item_tuple.description]□ contains "Bicycle"{
        □([<items.xml>.*.item_tuple]offered_by){
        □([<items.xml>.*.item_tuple]description){
        □(([<items.xml>.*.item_tuple]itemno){
        □([□]<items.xml>.*.item_tuple){items.xml}}}}})
        _[<items.xml>.*.item_tuple.offered_by]□ = [<users.xml>.*.user_tuple.userid]□
        (□[<users.xml>.*.user_tuple.rating]□ = "A"{
        □([<user.xml>.*.user_tuple]rating){
        □([<user.xml>.*.user_tuple]name){
        □([<user.xml>.*.user_tuple]userid){
        □([□][<user.xml>.*.user_tuple]){users.xml}}}}})}}}
```

Figure 12: A complex XQuery query and its algebraic representation

Optimizing our algebra, which operates over XML data, is similar to optimizing relational or object-oriented algebras. We are in the process of developing a cost-based optimizer based on the algebra. To do so, we will be using *OPT++*, a powerful optimization framework, which has been used to implement relational and OO systems [KD99]. It is possible to map our algebra to such an existing optimizer, since we only have to account for a few additional operators (*unnest* and the construction operators) and these operators have similar behavior to existing ones. For instance, *unnest* can be thought of as a more complex *structural selection* with node reachability acting as the selection predicate and path expression appearance estimation taking the role of selectivity statistics [AA+01].

Optimizers can be viewed as operating on three levels: expression transformation, access path selection, and algorithm selection. The latter two have to do with the way an expression is executed, that is, how the system will choose to access data (e.g., through *scans* or *indices*) and which algorithms it will employ to perform the operation (e.g., a *nested loops join* or a *hash join*). Since these latter two depend heavily on the details of an implementation framework, in this section we focus on implementation independent optimization by rewriting using equivalences.

The following list presents a collection of equivalence rules that can be proven to hold[4] over our algebraic framework:

1. $\square([C]F_1)\{\square([C]F_2)\} \equiv \square([C]F_2)\{\square([C]F_1)\}$
2. $\square([C_1]F_1)\{\square([C_2]F_2)\} \equiv \square([C_2]F_2)\{\square([C_1]F_1)\}$, if B_1 and B_2 do not share a common prefix.
3. $E\{\square([P]F)\{\square([B]P)\}\} \equiv E\{\square([B]P.F)\}$, if E is an expression and P is not used as an entry poing in E.
4. $\square_p\{\square(P)\} \equiv \square(P)\{\square_p\}$, if p does not operate on elements specified by P.
5. $\square_{p_1}(\square_{p_1}) \equiv \square_{p_1 \square p_2}$
6. $\square_{p_1}(A\__{p_2} B) \equiv A\__{p_1 \square p_2} B$
7. $\square_p(A\_B) \equiv (\square_p(A))\_B$, if p applies only to A.
8. $A\_B \equiv B\_A$, if the input collections to the join operation are anything but lists.
9. $A\_(B\_C) \equiv (A\_B)\_C$, if the input collections to the join operation are anything but lists.
10. $\square(P_1,P_2,\ldots)\{E_1\{E_2\}\} \equiv E_1\{\square(P_1,P_2,\ldots)\{E_2\}\}$, if E_1, E_2 are expressions and E_1 operates only on P_1, P_2,…
11. $E_1\{c_*(\ldots,[P_1, P_2, \ldots])\{E_2\}\} \equiv c_*(\ldots,[P_1, P_2, \ldots])\{E_1\{E_2\}\}$, if E_1, E_2 are expressions and c_* is any construction operator.

Rules 1 and 2 deal with *unnesting commutativity*, specifying under which circumstances it is possible to interchange the execution order of two unnesting operations, while Rule 3 deals with *unnesting elimination*, indicating the conditions under which it is possible to eliminate an *unnest* operator from the expression. Since path expressions have associated selectivities it might be better in some cases to execute one unnesting operation before another as such an action might change the size of the intermediate result set. Rule 4 deals with *selection/unnesting commutativity*, specifying the conditions under which it is possible. Again, an optimizer's decision to use this rule is based on the selectivities of the participating selection and unnesting operations. The next rule deals with *predicate factoring*, since in some cases it would be beneficial to factor two selection predicates into a single one, or to analyze them into two distinct ones. Rule 6 has to do with *selection elimination* with regards to a join operation. A selection predicate imposed on the result of a join can be pushed down into the join predicate, eliminating the selection operation. Rule 7 is the well-known *selection push-down* rule, that aims at evaluating selections as soon as possible, resulting in smaller intermediate result sizes. *Join associativity* and *join commutativity* motivate next two rules, which provide the basis for join enumeration. It should be noted here that for the two rules to hold, their inputs should be sets or bags, i.e., unordered collections. If their inputs are lists, the result will not be the same. Using

[4] Due to space constraints we will not present the proofs in this paper.

homomorphisms though, we can make sure the inputs to the join operations are unordered collections. Finally, the last two rules deal with the pushing-down of *projections* and *constructions*, presenting the cases in which such actions are possible.

Looking at the rules, one notices that they are not much different than the standard equivalence rules that hold over relational algebra, which lends credence to our belief that existing relational optimizer technology can be applied to our algebra with relatively few modifications. To provide further evidence, we decided to run some preliminary timing experiments in our prototype XML query processor using the equivalence rules of this Section to generate different execution plans. Our hardware setup consisted of a Pentium-III processor operating at 800 MHz with 256 MB of physical memory, running RedHat Linux 6.2.

5.1 Ordering Unnestings and Selections

In our first experiment, to see if any of our non-relational operators present opportunities for optimization, we explored queries using the *unnest* operator. We generated a simple query containing two *unnest* and two *select* operators over a simple dataset. The first unnest operator, $\square_1$, extracted a path expression with a small cardinality, while the selection, $\square_2$, had a rather high selectivity. On the other hand, the second unnest operator $\square_2$, had a large cardinality, while the selection, $\square_2$, had a low selectivity factor. Table 1 summarizes the setup.

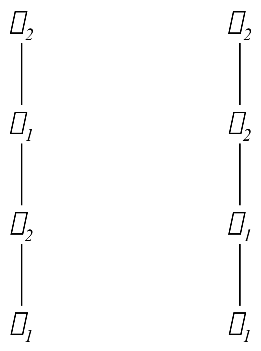
Operator	$\square_1$	$\square_1$	$\square_2$	$\square_2$
Selectivity	5	0.2	1091	0.5

Table 5: Setup for the unnesting and selection ordering experiments

We generated the plans in Figure 13 to see how the interaction between unnests and selects influences execution time. We expected the left plan of Figure 13 to be faster than the right plan, since it uses two select operators to limit the cardinality of the input to the unnest operators. Figure 14 shows that this was indeed the case. The conclusion is that an algebra that selection predicates and unnesting operations presents opportunities for optimization by reordering these operators. In fact, as Figure 14 shows, in this case a simple switch in execution order doubled performance.

5.2 Ordering Selections and Joins

In our second experiment, the objective was to see whether traditional join ordering optimization was useful with our algebra. We again used a simple query: there were three inputs *A*, *B* and *C* joined by two equi-join predicates. Moreover, we added two selection operators on *A.a* and *C.c*. The details of the documents and the various cardinalities and selectivities are given in Table 6. We organized the plan for the query in three different ways and compared their performance. Symmetric hash join [WA91] was used as the evaluation algorithm in all three cases. Figure 15, Figure 16 and Figure 17 present the execution plans, while Figure 18 presents the results.



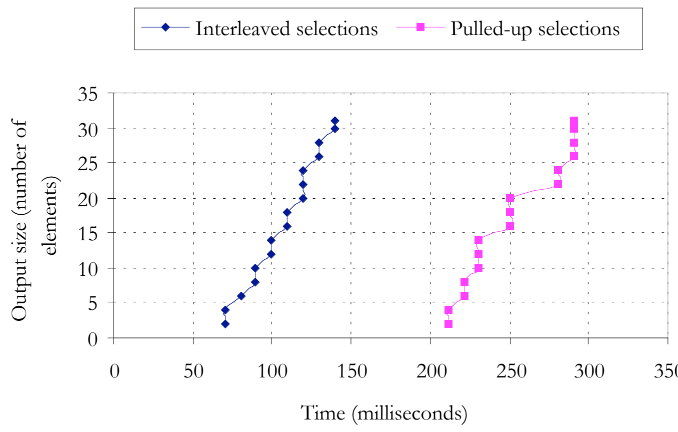


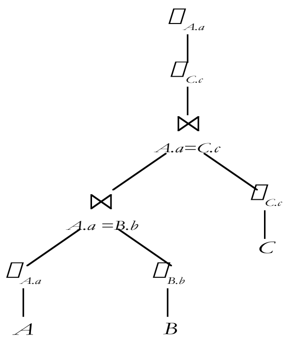
Figure 13: Pulling up selections (left) and interleaving selection and unnest operations (right)

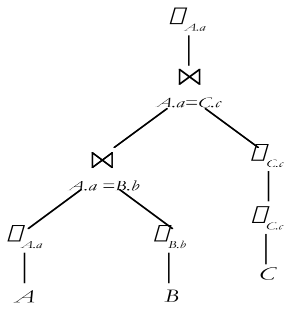
Figure 14: Performance results

Source	*# Elements*	*Selection predicate selectivity*	*Join predicate selectivity*
A	5000	□$p_{(A.a)}$= 0.2	□$_{(A.a=B.b)}$ = $2{\cdot}10^{-3}$
B	10000	N/A	N/A
C	20000	□$p_{(C.c)}$ = 0.25	□$_{(A.a=C.c)}$ = $5{\cdot}10^{-3}$

Table 6: Experimental setup

It is obvious that different plans for the same query can have substantially different execution times. Interestingly, employing traditional heuristics like pushing down selections is not always guaranteed to provide better execution times, since certain combinations of predicates and source sizes apparently lead to scenarios in which the heuristic may not work. For instance, the third plan of our experiments might be expected to have the best performance, however, the second plan that does not fully push down selections turns out to be better. We plan to address all these issues during the implementation of the optimizer.





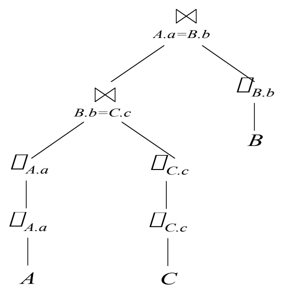


Figure 15: Selections are not pushed down

Figure 16: Selections partially pushed down

Figure 17: Selections fully pushed down

6 Related Work

Algebraic representations of queries have been extensively studied, dating back to Codd's definition of the relational model [C70], the introduction of the relational algebra, and the proof of its equivalence to tuple-oriented relational calculus [C72].

There has also been significant work on algebras in the context of object-oriented databases. Our work has been influenced mostly by [FM95a] and [FM95b]. Monoid homomorphisms are a strong candidate for an XML calculus we plan to devise in our future work.

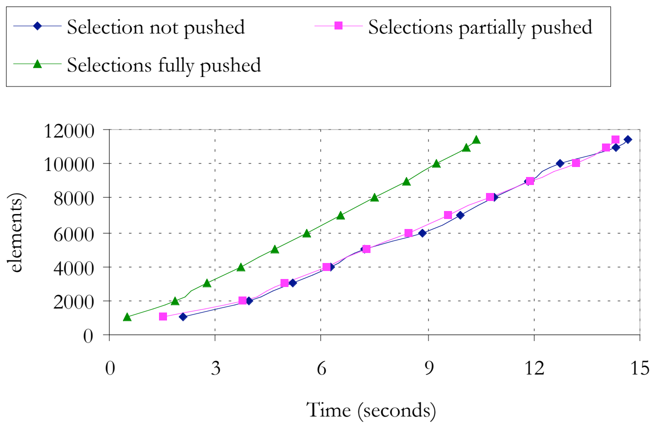


Figure 18: Performance of the three join ordering plans

The object-oriented algebra presented in [BM+93] uses a combination of *materialize* and *unnest* to access set-valued attributes, with *unnest* extracting references from sets and *materialize* resolving these references to in-memory objects, which is close to what our *unnest* operator does (Section 4.2.2). Our approach differs in two aspects: (i) It is concerned with data exploration, while [BM+93]'s *unnest* operates on a known structure (set-valued attributes). In our framework, there is no notion of a schema. We could easily unnest a path that leads to a single element. (ii) It has no notion of low-level operations such as memory reference resolution (as in the *materialize* case).

Marginally related to our work is the work of [LM+93], in which they present a strong case towards a variable (hence, binding) free algebra, designed to simplify rule formulation. Their point is that algebras using bindings result in difficult-to-optimize expressions.

The problem of querying over XML data has mainly been dealt with in the context of specific query language proposals. XML-QL [DF+98] was perhaps the first well-known XML query language, while the World Wide Web Consortium (W3C) has an evolving recommendation for XQuery [CF+01]. XSL [AB+00] and XPath [CD99] have been proposed as ways to restructure, present and access parts of XML documents. Studies in *semi-structured* data [A97], [B97] are also applicable to querying XML data. The focus of the *Lore* [MA+97] project was on querying semi-structured data. They have proposed a framework for XML query representation, optimization and execution [MW99b]. Their work and other studies in formal data models for XML querying [BMR99a], [FS+] have also substantially influenced our work.

The TAX algebra [JL+01] is probably the closest one to our work. It is a tree algebra that takes a different approach to querying XML data. Instead of mapping their queries to flat structures, they create *annotated patterns*, which they impose over the XML data, identifying points of the patterns' occurrences. The patterns then act as *copy constructors* of new XML data, containing only the matched portions of the original data. Our approach differs in two dimensions: firstly, we do not make any copies of XML data; we handle that implicitly through generating and manipulating context. Secondly, we do not use patterns to identify XML structures; such actions are explicitly captured through our *unnest* operator and the functions and predicates frameworks.

Finally, the World Wide Web Consortium has proposed a formal semantics for XML [FF+01], with a strong inclination towards type inference.

7 Conclusions and Future Work

We have presented an algebraic framework for XML query representation. The basis for this framework is the notion of context. Our algebra is powerful enough to be the target algebra for the "FLWR" and "IF-THEN-ELSE" constructs of the XQuery language (for those who are interested, we present a sketch of an algorithm to translate from the XQuery "FLWR" expressions to our algebra in the Appendix). We believe that our use of the context construct instead of the more common bindings will make it easier to construct efficient evaluators and powerful optimizers for the algebra.

We have already implemented this framework in the context of our XML query processor, which uses our algebra to create logical execution plans. These plans are later translated into physical plans, using ad-hoc heuristics. The next step into that direction is to build a working cost-based optimizer on top of the algebra. This will further verify our intuition that relational optimization principles with a few additions can be used in the context of XML query processing systems.

Acknowledgments: Omitted for anonymity.

References

[A97] S. Abiteboul, Querying Semi-structured Data, Proceedings of the International Conference on Database Theory, Delphi, Greece, January 1997, pp. 1-18.

[AA+01] A. Aboulnaga, A. R. Alameldeen, J. F. Naughton, Estimating the Selectivity of XML Path Expressions for Internet Scale Applications, Proceedings of VLDB 2001, Rome, Italy, September 2001.

[AB+00] S. Adler, A. Berglund, J. Caruso, S. Deach, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles, Extensible Stylesheet Language (XSL), W3C Candidate Recommendation, http://www.w3.org/TR/xsl, November 2000.

[AU79] A. V. Aho and J. D. Ullman, Universality of Data Retrieval Languages, Principles of Programming Languages, 1979, pp. 110-120.

[B97] P. Buneman, Semi-structured Data, Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, AZ, May 1997.

[BM+93] J. A. Blakeley, W. J. McKenna, G. Graefe, P. Buneman and S. Jajodia, Experiences Building the Open OODB Query Optimizer, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993 pp. 287-296.

[BMR99a] D. Beech, A. Malhotra and Michael Rys, A Formal Data Model and Algebra for XML, W3C XML Query working group note, September 1999.

[BMR99b] D. Beech, A. Malhotra and Michael Rys, A Formal Data Model and Algebra for XML, http://www-db.stanford.edu/dbseminar/Archive/FallY99/malhotra-slides/tsld001.htm, 1999.

[BPM98] T. Bray, J. Paoli and C-Sperberg-McQueen, Extensible Markup Language (XML) 1.0, W3C Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210, February 1998.

[C70] E. F. Codd, A Relational Model for Large Shared Data Banks, Communications of the ACM, 13(6):377-387, June 1970.

[C72] E. F. Codd, Relational Completeness of Database Sublanguages, R. Rustin (editor), Data Base Systems, Prentice-Hall, 1972, pp. 65—98.

[CA+94] V. Christophides, S. Abiteboul, S. Cluet and M. Scholl, From Structured Documents to Novel Query Facilities, Proceedings of the ACM SIGMOD Conference, Minneapolis, MN, May 1994.

[CB+00] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda and F. Velez, Object Data Standard ODMG 3.0, Morgan-Kaufman, January 2000.

[CCB94] C. L. A. Clarke, G. V. Cormack and F. J. Burkowski, An Algebra for Structured Text Search and A Framework for its Implementation, Technical Report CS-94-30, Dept. of Computer Science, University of Waterloo, Waterloo, Canada, August 1994.

[CD+00] J. Chen, and D. J. DeWitt, F. Tian and Y. Wang, Niagara-CQ: A Scalable Continuous Query System for Internet Databases, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, May 2000, pp. 379-390.

[CD99] J. Clark and S. DeRose, XML Path Language (XPath), Version 1.0, W3C Recommendation, http://www.w3.org/TR/xpath, November 1999.

[CF+01] D. Chamberlin, D. Florescu, J. Robie, J. Simeon and M. Stefanescu, XQuery: A Query Language for XML, W3C Working Draft, http://www.w3.org/TR/xquery, June 2001.

[DF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, A Query Language for XML, Submission to W3C available at http://www.w3.org/TR/NOTE-xml-ql, August 1998.

[FF+01] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon and P. Wadler, XQuery Formal Semantics, W3C Working Draft, http://www.w3.org/TR/2001/WD-query-algebra-20010215, February, 2001.

[FM95a] L. Fegaras and D. Maier, Towards an Effective Calculus for Object Query Languages, Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995, pp. 47-58.

[FM95b] L. Fegaras and D. Maier, An Algebraic Framework for Physical OODB Design, Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995.

[FS+] M. Fernandez, J. Simeon, D. Suciu and P. Wadler, A Data Model and Algebra for XML Query, http://www.cs.bell-labs.com/wadler/topics/xml.html#algebra.

[JL+01] H. V. Jaggadish, L. V. Lakshmanan, D. Srivastava and K. Thompson, TAX: A Tree Algebra for XML, Proceedings of DBPL 2001.

[KD99] N. Kabra and D. J. DeWitt, OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization, VLDB Journal, 8(1):55-78, 1999.

[LM+93] T. Leung, G. Mitchell, H. Subramanian, B. Vance, S. L. Vanderberg, S. B. Zdonik: The AQUA Model and Algebra, Proceecings of DBPL 1993.

[MA+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass and J. Widom, Lore: A Database Management System for Semistructured Data, SIGMOD Record, 26(3):54-66, September 1997.

[Mai] D. Maier, Database Desiderata for an XML Query Language, http://www.w3.org/TandS/QL/QL98/-pp/maier.html.

[MW99a] J. McHugh and J. Widom, Optimizing Branching Path Expressions, Technical Report, Stanford University, June 1999.

[MW99b] J. McHugh and J. Widom, Query Optimization for XML, Proceedings of the 25th VLDB Conference, Morgan-Kaufmann, Septempber 1999, pp. 315-326.

[ND+] J. F. Naughton, D. J. DeWitt and D. Maier et al, The Niagara Internet Query System, Submitted for Publication.

[SS+00] J. Shanmugasundaram, E. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, B. Reinwald, Efficiently Publishing Relational Data as XML Documents. In *Proc of VLDB 2000,* Cairo, Egypt, August 2000.

[ST+00] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton and D. Maier, Architecting a Network Query Engine for Producing Partial Results, WebDB 2000, Dallas, TX, May 2000.

[T55] A. Tarksi, A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific Journal of Mathematics, 5(2):285-309, 1955.

[TD+] F. Tian, D. J. DeWitt, J. Chen and C. Zhang, The Design and Performance Evaluation of Various XML Storage Strategies, Submitted for publication.

[W3C] World Wide Web Consortium, http://www.w3c.org.

[WA91] A. N. Wilschut and P. M. G. Apers, Pipelining in Query Execution, Conference on Datbases, Parallel Architectures and their Applications, Miami, FL, 1991.

Appendix: Translation Algorithm

This appendix presents a translation algorithm from XQuery to our algebra. The translation algorithm covers the most common expressions of XQuery, namely the *FLWR* expressions as well as the more procedural *IF-THEN-ELSE* expressions. Some additional bookkeeping structures and functions needed for the algorithm to work correctly are presented in Table 7.

The algorithm traverses the XQuery query, identifying parts of correspondence to the retrieval algebraic operators, which are inserted into the appropriate *query branch*. Each branch acts as a sub-expression of the resulting final expression. XQuery uses XPaths and the latter have a way of tangling retrieval operations along with filtering ones, so a single XQuery pattern retrieval construct can result in a more complex algebraic expression. Once all the retrieval operators are handled, the next step is to handle all groupings appearing in the query's *LET*-clause, again inserting them into the appropriate branch.

After retrieval patterns and groupings are accounted for, the *WHERE*-clause is handled. One can do an optimization at this point, and treat predicates pertinent to a single branch before predicates operating on two branches (which result in join operations). That way selection operations are pushed down, and then the algorithm can move on to combine the multiple branches into a single expression by treating join predicates or, if there are no predicates joining the branches, by creating Cartesian products. The result will be a single expression for the whole query. At this point the algorithm starts constructing the final output (the *RETURN*-clause): Appropriate constructors are inserted into the expression, and finally a projection takes place to expose the final output element. Algorithm 3 presents a high-level sketch of the algorithm showing how the various XQuery constructs (XPaths and clauses) map to specific operations of our algebra.

Structure or	*Description*

Function Name	
Branches	A list containing pairs of an individual branch of the final algebraic expression and a handle to it, *branch*$_{id}$. These branches will be combined into a single expression at the end of the translation phase.
Bindings	A table of (*binding*, *path expression*, *branch*$_{id}$) triplets. Used for look-ups when creating the regular path expressions the algebra uses.
lookup$_{path}$*(var)*	Looks up *var* inside Bindings and returns its path expression.
lookup$_{branch}$*(var)*	Looks up *var* inside Bindings, obtains its *branch*$_{id}$ and uses it to fetch the corresponding algebraic expression.

Table 7: Bookkeeping structures and functions for the translation algorithm

translate(Q): *Translation from XQuery Q to an algebraic expression*

XPaths

$a_1/a_2/\ldots/a_n$ □ $[□]a_1.a_2.□\ .a_n$

$a_1//a_2/\ldots/a_n$ □ $[□]a_1.*.a_2.□\ .a_n$

$a_1/a_2/\ldots/@a_n$ □ $[□]a_1.a_2.□\ .@a_n$

$a_1/a_2/\ldots/a_n[m]$ □ $[□]a_1.a_2.□\ .a_n[m]$

FOR-clause

\$var IN `doc()`$/a_1/a_2/□\ /a_n$ □

create binding for *\$var*, create new branch, □([□]`doc()`$.a_1.a_2.□\ .a_n$)

\$var$_1$ IN *\$var*$_2/a_1/a_2/□\ /a_n$ □

create binding for *\$var*$_1$, P □ *lookup*$_{path}$(*\$var*$_2$), □([P]`doc()`$.a_1.a_2.□\ .a_n$)

\$var IN `doc()`$/a_1/a_2/□\ /a_{n-1}[p(a_n)]/a_{n+1}/□$ □

create binding for *\$var*, create new branch,

□([`doc()`$.a_1.□\ .a_{n-1}]a_{n+1}.□$)(□$_{p([\text{doc()}.a_1.a_2.\ldots.a_{n□1}]a_n)}$ (□([□]`doc()`$.a_1.□\ a_{n-1}$)))

\$var$_1$ IN *\$var*$_2/a_1/a_2/□\ /a_{n-1}[p(a_n)]/a_{n+1}/□$ □

create binding for *\$var*$_1$, P □ *lookup*$_{path}$(*\$var*$_2$),

□($[P.a_1.□\ .a_{n-1}]a_{n+1}.□$)(□$_{p([P.a_1.a_2.\ldots.a_{n□1}]a_n)}$ (□([P]$a_1.□\ a_{n-1}$)))

\$var$_1$ IN P, *\$var*$_2$ IN P { *P refers to document d* }

create binding for *\$var*$_1$, create binding for *\$var*$_2$, create new branch,

□(P_1)(□(d,d_1)) $—_{id(P1)=id(P2)}$ □(P_2)(□(d,d_2))

\$var$_1/P_1$ BEFORE | AFTER *\$var*$_2/P_2$

□(*id*([*lookup*$_{path}$(*\$var*$_1$)]P_1) < | > *id*([*lookup*$_{path}$(*\$var*$_2$)]P_2)))

LET-clause

\$var = `doc()`$/a_1/□\ /a_n$ □

create binding for *\$var*, □([□]`doc()`$.a_1.□\ .a_n$)

\$var$_1$ = *\$var*$_2/a_1/a_2/□\ /a_n$ □

create binding for *\$var*$_1$, P □ *lookup*$_{path}$(*\$var*$_2$),

B □ *lookup*$_{branch}$(*\$var*$_2$), □([P]$a_1.□\ a_n$)(B)

WHERE-clause

\$var$/P$ = *const* □

□$_{lookup_{branch}}$(*\$var*) = *const* (*lookup*$_{branch}$(*\$var*))

\$*var*$_1/P_1$ = *\$var*$_2/P_2$ □

PP_1 □ [*lookup*$_{branch}$(*\$var*$_1$)]P_1, PP_2 □ [*lookup*$_{branch}$(*\$var*$_1$)]P_1

lookup$_{branch}$(*\$var*$_1$) $—_{PP_1=PP_2}$ *lookup*$_{branch}$(*\$var*$_2$)

RETURN-clause

`text` □ c_d(*text*)

`<tag>` *\$var* `</tag>` □ c_e(`tag`, [], [*lookup*$_{path}$(*\$var*)])

```
        <tag attr=$\$var_1$> $\$var_2$ </tag> □
                $c_e$(tag, [attr], [$lookup_{path}(\$var_2)$])($c_a$(attr, $lookup_{path}(\$var_1)$))
        <tag> $structure$ </tag> □
                $E$ □ $translate(structure)$
                if (already within a FLWR expression) then $E$ □ □($value(tag)$) $E$
                $c_e$(tag, [], [$tag(structure)$]) [$E$]
IF-THEN-ELSE-clause
        if $predicate$ then $statements_1$ else $statements_2$ □
                □$_{predicate}(translate(statements_1))$ □ □$_{□predicate}(translate(statements_2))$
No more clauses
        while there is more than one branch do
                $B_1$ = first branch, $B_2$ = second branch,
                create branch $B_1$ □ $B_2$, delete $B_1$, delete $B_2$
        end while
```

Algorithm 3: Translation algorithm from XQuery to an algebraic expression