Paper Number 363

Following the paths of XML Data: An algebraic framework for XML query evaluation

Abstract

This paper introduces an algebraic framework for expressing and evaluating queries over XML data. It presents the underlying assumptions of the framework, describes the input and output of the algebraic operators, and defines these operators and their semantics. It evaluates the framework with regard to other proposed XML query algebras. Examples show that this framework is flexible enough to capture queries expressed in Quilt, one of the dominant XML query languages. We have used this algebra in the context of an Internet query engine, in which it is used to formulate logical plans for XML-QL queries. We define equivalence rules that provide opportunities for optimization, and give example cases that point out the usefulness of these rules.

1 Introduction

Recently, the database community has focussed a lot of attention on mechanisms for querying XML. The result has been a number of XML query languages, and a number of prototype implementation of these query languages. However, there is to date no accepted "XML query algebra" that can serve as a well-understood, language-independent intermediate representation for XML queries. This is in marked contrast to relational systems, in which relational algebra has long been recognized as useful intermediate form in the process of query execution.

Almost 30 years of experience with relational algebra and relational systems have demonstrated that a unified, declarative, and query language-independent way of formulating queries provides several important optimization opportunities, including:

- The opportunity to introduce access methods other than opening and scanning XML documents;
- The opportunity to use cost-based optimization along with query rewriting through equivalence transformations.

An XML query system must also be able to make use of these opportunities, or it will be doomed to execute ad-hoc, suboptimal plans.

The contribution of this paper is the introduction of a formal query algebra over XML data. A basic premise of the algebra is that the XML data is *semi-structured* in the sense that it *does* have some structure,

but this structure may not be known before the query execution begins. Because of this, the algebraic framework explores the structure of the data as well as filtering and transforming the data. Our proposed algebra has implemented in the context of a research project[1] that currently uses an extended version of XML-QL [DFF+98] as its query language. Using the same algebraic framework, we are modifying our system to support Quilt [CRF00]. The goal of the project is to build an Internet query engine by combining XML query processing with text-in-context search engine functionality, treating the search engine as a powerful access method. Using our proposed algebraic representation of a query, one can extract complex text-in-context retrieval operations and issue them directly to the search engine, retrieving only relevant data, using the algebraic operators to further filter these data.

The rest of the paper is structured as follows. Section 2 presents the related work. Section 3 further motivates the need for a new algebra by pointing out the differences between the requirements of an XML query engine and traditional relational and object-oriented query engines. Section 4 introduces the XML data model and looks at the problems existing algebras encounter with this new model. This Section also proposes a novel way to treat operator input and output that overcomes these problems. The algebraic operators are defined in Section 5, while advanced features of the algebra are presented in Section 6. Section 7 demonstrates the algebra's usability by providing examples of algebraic expressions for queries expressed in Quilt. Section 8 presents and motivates equivalence rules, which form the foundation of a query optimization framework. Advanced features of the algebra are presented in Section 6. Finally, Section 9 summarizes the framework and identifies our planned future directions.

2 Related Work

Algebraic representations of queries have been extensively studied, dating back to Codd's definition of the relational model [Cod72] the introduction of the relational algebra, and the proof of its equivalence to tuple-oriented relational calculus [Cod70].

There has also been a lot of 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 that we plan to devise in our future work (see Section 9).

The object-oriented algebra presented in [BMG93] uses a combination of *materialize* and *unnest* to access set valued attributes which is close to what our *Follow* operator does (Section 5.2). Unnest extracts references from sets and materialize resolves these references to in-memory objects. Our approach differs in two aspects:

1. It centers on data discovery, while *unnest* operates on a known structure (set-valued attributes).
2. It has no notion of low level operations such as memory reference resolution (as in the *materialize* case).

[1] The name of the project will not be disclosed for anonymity purposes.

The problem of querying over XML data has mainly been dealt with in the context of specific query language proposals. XML-QL [DFF+98] was perhaps the first well-known XML query language. Quilt [CRF00] is a recently proposed language that is rapidly growing in popularity. XSL [ABC+00] and XPath [CD99] have been proposed as ways to restructure, present and access parts of XML documents. Studies in *semi-structured* data [Abi97, Bun97] are also applicable to querying XML data.

The focus of the *Lore* [MAG+97] project is queries over semi-structured data. They have proposed a framework for XML query representation, optimization and execution [MW99b]. This work and other studies in formal data models for XML querying [BMR99a, FSSW, FSSW] has also substantially influenced our work. However, based upon our experiences with implementing an algebra for XML querying, none of these proposals really solves the XML algebra problem. We discuss this more fully in Section 4.2.

3 The Need for a new Algebra

To define our operations framework, we begin by closely examining the desired functionality. The World Wide Web Consortium [W3C] poses three basic requirements that any XML data manipulation mechanism must satisfy:

1. **Pattern retrieval:** given an XML pattern and a collection of XML data, the mechanism should be able to identify positions in the collection where this pattern appears.

2. **Filtering:** once patterns have been identified, they should be further selected based upon the values of their specific attributes.

3. **XML Construction:** to support document transformations, any XML data manipulation mechanism must provide a way to create new data from existing data.

So, where should we turn to formulate an algebra to satisfy these requirements?

One logical place to look is in the proposals for algebras in object-oriented systems. Object-oriented systems certainly implement pattern retrieval operations, since their use of collections and sets of arbitrary nesting is a way of formulating patterns. Furthermore, path expressions are a way to navigate through these patterns. These capabilities have been captured in a number of object algebras and calculi, each with their own strengths and weaknesses. Unfortunately, this superficial similarity between object-oriented systems and XML data does not go deep enough to solve the whole problem. The basic issue, as we describe later in this section, is that with object-oriented systems, query processors have at their disposal a lot of structural information from the database class definitions. This information is not always available when querying XML data sets.

Another logical starting point is relational systems. Relational systems have used algebras to efficiently perform *filtering* operations for the past thirty years. However, these algebras do not directly support some new *text-in-context* search operations, such as containment, distance, direct containment and so forth, that are extremely useful in XML querying. This is probably not a serious deficiency, as relational filtering operators can be extended to provide support for a greater collection of predicates. The more serious problem

is that, like their object-oriented counterparts, relational algebras assume a great deal of knowledge about the structure of the data over which they are querying.

Neither relational nor object-oriented algebras provide sufficient support for document transformations. A query over relational data always produces a relation, which has a pre-defined, flat structure. Similarly, queries over object-oriented data produce collections of objects and do not allow manipulation of the class hierarchy as part of the query. None of these paradigms contains the notion of providing arbitrarily structured output, which is essential for performing document transformations. Consequently, an XML algebra will need specialized operators that can do this arbitrary restructuring.

We now turn to what we consider the key issue in constructing an XML algebra: the lack of information about the structure of the data. XML data have no notion of a schema in the sense that relational or object-oriented systems have schemas. The closest concept to a schema is perhaps DTDs or XML-Schemas, but both of these are optional. Furthermore, there is a growing need to query over related but heterogeneous collections of XML data. In these applications, each document may conform to some schema, but different documents in the collection correspond to different schemas. If a query processor is to be able to issue queries over such a diverse set of documents, it cannot make assumptions such as "the documents all conform to a known schema."

In the most general case, the only information we have when we pose queries over an XML document is that the contents of the document conform to the XML data model (see Section 4.1). Consequently, handling filtering operations becomes more difficult because we are not certain that the attributes of the data on which we wish to filter actually exist in a given document! Therefore, an XML algebra must be able to explore the data. As a result, the first two requirements for an XML querying mechanism, pattern retrieval and filtering, are collapsed into a single one: querying on both structure *and* content.

4 Querying XML Data

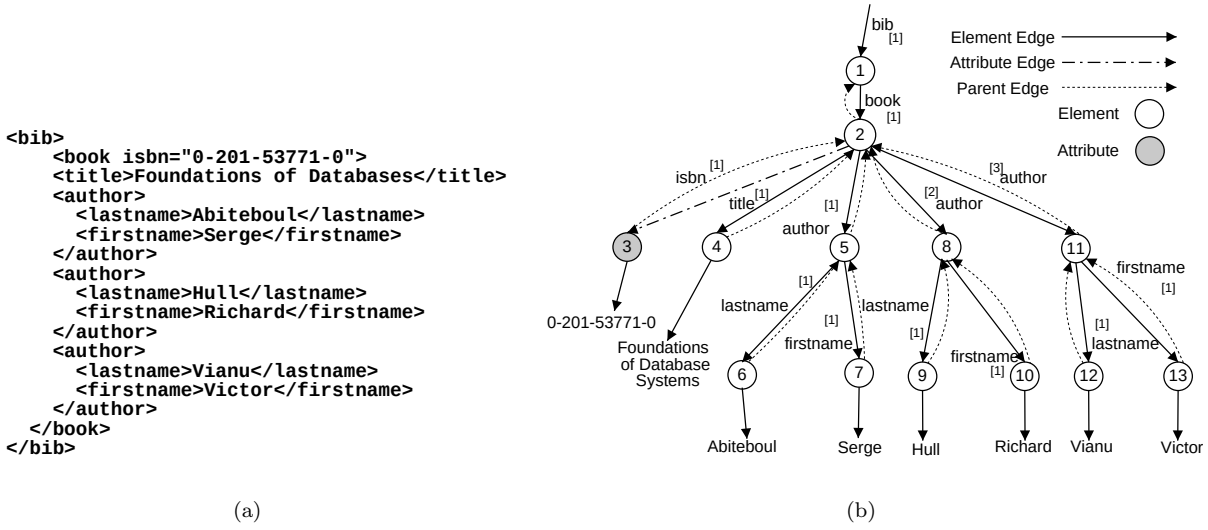
In this section we review the XML data model and concentrate on existing approaches to formulate an algebraic framework for this model. We will identify problems with these approaches and propose our solution. We think that the key to solving these approaches lies in the details of the definitions of operator input and output; these definitions must be rich enough to both capture the history of how an input was created and to pass along enough information to support subsequent operations.

4.1 The XML Data Model

Every XML documents adheres to a data model defined in the standard XML specification [BPCSM98]. The model is represented by a directed rooted graph. Figure 1(a) shows an example XML document, in which there is a topmost, unique *element*, `bib`, known as the root of the document. All elements are enclosed into the topmost element, `bib`, known as the root of the document. Three `book` sub-elements reside within the root. This nesting of sub-elements can go on to an arbitrary level. It is possible to have *attributes* associated with elements, as is the case with the `isbn` attribute of the `book` elements.

Figure 1(b) depicts the corresponding graph model for the document in Figure 1(a). Elements and attributes correspond to vertices in the graph. Directed, named edges connect the vertices, with the tag of the corresponding element or the attribute name acting as the name 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 points to another arbitrary element, potentially leading to cyclic structures in the graph. Note that parent edges appear only between vertices that are connected with element or attribute edges. There is a total ordering on vertices, which corresponds to a depth-first traversal of the graph. There is also an ordering between sibling edges.

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).



(a) (b)

Figure 1: An XML document fragment and its corresponding graph

4.2 Existing approaches and their problems

In the process of defining an algebra for an XML query engine, two studies served as our starting point: [BMR99a] and [MAG+97]. As our implementation of an XML query engine progressed, we identified significant shortcomings of both of these approaches, which led us to decide to define a new algebra. The problems we faced are summarized below:

Schema information. Previous proposals for algebras require information on the structure of the XML document being queried. For example,[MAG+97, MW99a] are heavily dependent on the existence of *data guides*, which provide *a priori* information on the interconnections between the graph's vertices.

Even though data guides are dynamic, they need to be consulted at query compile time. Our algebra does not rely on an underlying schema.

The algebraic framework of [BMR99a] assumes that at operator creation time, the type of each vertex (attribute, element or reference) is known. For instance, when navigating to an attribute vertex, the path expression employed to reach the target vertex has to explicitly state the fact that the vertex corresponds to an attribute. In the absence of schema information, requiring knowledge of the vertices' types limits exploration possibilities. This is not the case in our algebra since the fact that a vertex may correspond to an attribute, an element or a reference is transparent to the path expression. Explicit specification of the destination vertex's type, however, is possible (see Section 4.3).

Higher complexity. [BMR99a] defines a large collection of operators and explicitly preserves all variable bindings appearing in a query, by creating multiple clauses per query and assigning intermediate results to these bindings. This leads to complex structures when representing a query, which makes the operators hard to deal with in an implementation.

Our algebra, because of its powerful definitions for operator input and output can express all queries into a single expression. Moreover, by using the *entry point notation* for path expressions (Section 4.3), the number of operators used in a single query is reduced by avoiding inverse navigation. In contrast [BMR99a] make heavy use of inverse navigation.

Our definitions of operator input and output also help us avoid approaches such as the one in [MW99b], in which each edge traversal in the XML graph is represented by a different operator (called *Discover*). Once all traversals are accounted for, new operators (called *Chain*) are used to ensure the path expression structure remains intact. For instance, after retrieving vertices corresponding to `author` elements and vertices corresponding to `book` elements, they have to ensure that the `author` elements reside inside `book` elements (i.e., `book`s are *Chain*ed with `author`s). An operator acting on the *structure* and not the *value* of the data has to be employed. We believe this is a problem, since the algebraic expressions should concentrate on value-oriented and not structure-oriented operations. If not, the algebra will once again produce more complicated expressions that are difficult to optimize and evaluate.

Handling of Joins. [BMR99a] propose that when joining multiple sources, a reference edge is created from one joining element to the other. We found this to be problematic for the following reasons:

- It provides no room for traditional database optimization techniques such as join ordering. A reference edge is directed, which means that $A \bowtie B$ is no longer the same as $B \bowtie A$.
- Although the added edge was not there in the original data, once it is added all subsequent operations can move along that edge. To avoid this, filtering operators should declaratively define operations on data and not change the data.

Use of XPath. A revision [BMR99b] of the algebra defined in [BMR99a] builds on the premises of the XPath [CD99] implementation. While XPaths may be a valuable language construct, in an algebra XPaths put too much complexity on the mechanism used to specify the vertex one wants to reach, as they support selections and generic filtering operations as part of their notation. We wanted our

vertex specification mechanism to be merely a way of reaching the data and not a way of performing any operations. All operations should be captured by the corresponding algebraic operators (rather than being buried in vertex specification.)

4.3 Proposed Solution

Having identified the problems of Section 4.2 through our (sometimes frustrating!) experience in implementing an XML query processing system, we set out as goals for our algebra that it:

- Be independent of schema information [Mai].
- Query on both structure and content.
- Generate simple, flexible, yet powerful algebraic expressions.
- Allow re-use of traditional optimization techniques (e.g., join ordering) and the introduction of new ones.

As we have stated earlier, when querying XML data, the fact that the data is *semi-structured* means that operators have the additional overhead of exploring the graph as well as filtering it. A comparison to the relational paradigm makes the observation more obvious. In relational algebra, all operators operate on sets of tuples. Once an operator accepts a tuple in its input, the tuple plus the schema of the tables being queried give all information the operator needs to operate on that tuple. Each tuple is *self-contained* in that sense.

In XML, there is nothing equivalent to a self-contained structure. One might think that a reasonable candidate for a self-contained XML structure is the element. If elements were self contained, then all operators would simply operate on *sets of vertices* (since vertices are what elements correspond to in the data model). Unfortunately, it is not that simple; the XML data model is what rules out that approach. Consider the following scenario, as illustrated in Figure 2: The set of vertices being operated on is a set of `book` elements. Suppose we apply an operator that filters these elements based on the content value of their `author` sub-element, and that an element qualifies if the value of *any* of its sub-elements satisfies the qualification. The output will then consist of `book` elements that may contain `author` sub-elements that do not satisfy the qualification. If at a later time one wishes to change the operation set to a set of `author` elements by following all relevant edges, one will end up having elements of the set that do not satisfy the already evaluated qualification. To recap: we filtered out books based upon a specific author; but then when we returned to consider authors of books, the fact that we had "filtered out" a specific author was lost, since all authors "came along" with the book element.

Our solution to this problem is the following: instead of having the operators operate on a set of vertices, we have them operate on a *set of sets of vertices*. Each set inside the set of sets contains the vertex being operated upon, as well as all the vertices already visited along the path to that vertex. Figure 3 shows how this solution rectifies the problem of Figure 2.

Since vertices are reachable by multiple path expressions (see Section 4.1), we have to revise the notion of having a set of elements. To allow for duplicates, we let all operators operate on a *collection of bags of*

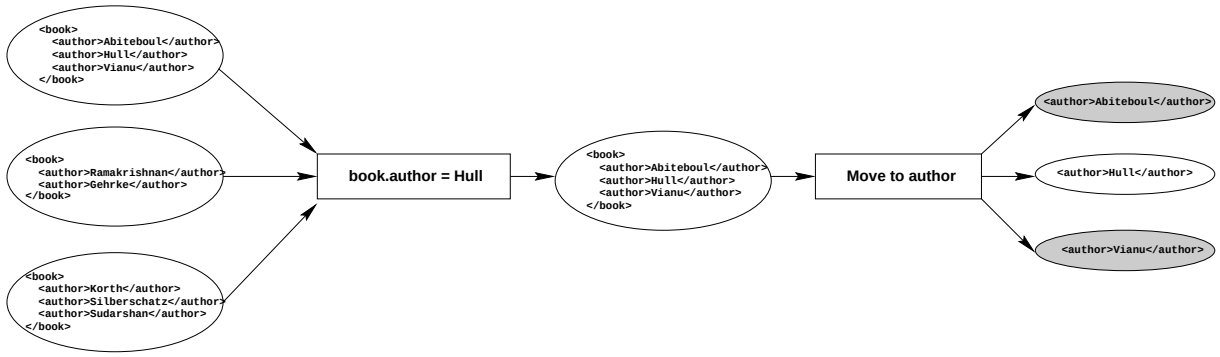


Figure 2: A problematic situation in using sets of elements as the basis for the algebraic operators. The shaded elements do not satisfy the already imposed condition, therefore they should not appear in the result set.

vertices. Moreover, there is no ordering within the bags; any permutation of elements within a bag leads to an equivalent bag.

What remains is a way to distinguish between the different elements of a bag. We accomplish that by annotating a bag element with the path expression that corresponds to the content of each element. For instance, for the document in Figure 1(a), assuming that each bag contains `book`, `author` and `title` elements (as well as the root of the document, which is always visited), the resulting bag will be the one in Figure 4. The annotation may be enriched with any variable bindings pertaining to the vertex specified by the path expression[2]. Moreover, we can have the same entry point annotations for different elements of the bags (as is the case with the Group operator – see Section 5.8).

To trigger an evaluation of a path expression on a specific element of a bag, our algebra uses the bag element annotations as entry points. For instance, suppose the actual content of the `author` element consists of two sub-elements: `lastname` and `firstname`. If we want to evaluate the absolute path expression `book.author.lastname` on the bag of Figure 4, we can do that on two elements of the bag, namely `book` and `book.author`. To explicitly specify which element of the bag should be used for the evaluation, we use the *entry-point notation* for path expressions. If the objective is to evaluate `book.author.lastname` inside `book`, we specify the path expression to be evaluated as `author.lastname:book`. The path expression is broken into two parts:

- A *relative forward part*: `author.lastname`.
- An *entry point*: `book`.

If we wished to evaluate the path expression inside the vertices reachable by the `book.author` path expression, then the path expression to be evaluated should be specified as `lastname:book.author`.

Path expressions can be made more elaborate by specifying the destination type of a vertex. For instance, to reach only attribute vertices one can use a different notation and qualify the edge name with the symbol

[2]These bindings can appear in the issued query, in cases such as the `ELEMENT_AS` and `CONTENT_AS` constructs of XML-QL [DFF+98], or the `FOR` and `LET` constructs of Quilt [CRF00].

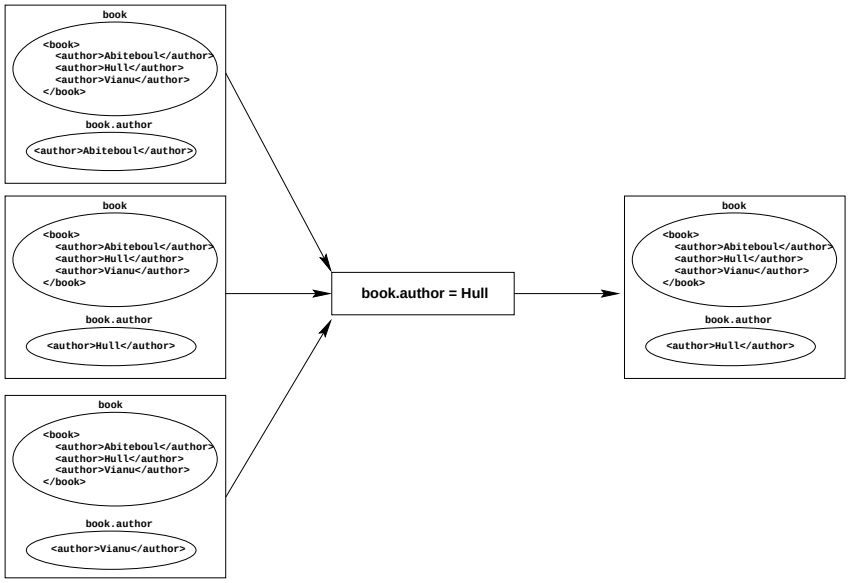


Figure 3: Using *collections of bags of vertices*, the problematic situation of Figure 2 is rectified, since there is access to both the original element *and* the qualifying sub-element (for brevity we show only part of the input).

[$Root_{\texttt{books.xml}}$, `book`, `book.title`, `book.author`]

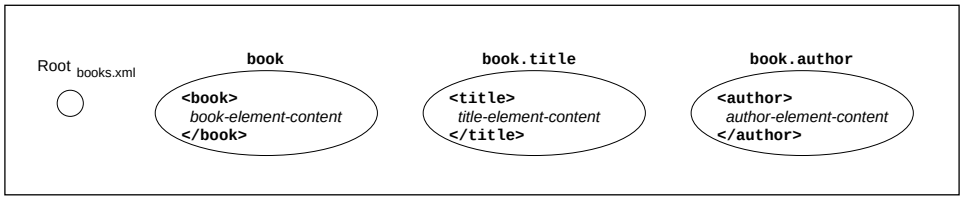


Figure 4: A bag of vertices and its path expression annotation

`@`. To specify element vertices, one can use the `#` qualifier while reference edges are accessible through the `->` qualifier. This makes it possible to distinguish between edges that have the same name but lead to different types of vertices. For example, suppose we face the following XML element:

```
<faculty title="Distinguished Fellow">
  <name>
    <lastname>Smith</lastname>
    <firstname>John</firstname>
  </name>
  <title>Professor</title>
</faculty>
```

Then, `faculty.@title` denotes the `title` *attribute*, while `faculty.#title` denotes the `title` *element*. The use of type qualifiers on path expressions, however, can be omitted, in which case the path expression denotes all reachable vertices regardless of their type. In the previous example, `faculty.title` will reach both the attribute and the element vertices.

Another navigational convenience is the *parent* construct, denoted by `^`, which, for any vertex, returns its parent vertex. For instance, `lastname^` reaches the parent of a `lastname` vertex, while `department.*.lastname^` reaches the parent vertex of `lastname` elements appearing at any depth under `department` elements.

Finally, relative ordering can be captured by path expressions. For instance, suppose we have XML data that corresponds to a book publication, and we want to access the third chapter of that book. By specifying a path expression such as `book.chapter[3]` we only operate on the third reachable chapter vertex.

5 Operators

Given the operator inputs and outputs as defined in the previous section, our next step is to introduce the actual operators. All operators, unless otherwise stated, as Section 6 presents, operate on the *value* of the XML data. These operators retrieve patterns, filter data and construct new XML.

5.1 Source (s)

Our first operator is the *Source* operator, denoted by the symbol s. It takes as its input parameter a list of documents that should be used as inputs. These documents could be specified in any number of ways, ranging from explicit lists to general wild card expressions to restrictions to sets of documents conforming to a DTD or XML-Schema. Table 1 presents some potential forms a system could use to specify the list of input documents. The output of the *Source* operator is a collection of singleton bags, each bag containing the unique root of the XML documents that satisfy the operator's criteria.

Form	Semantics
s(`*`)	All known XML documents
s(`bib*.xml`)	All XML documents whose filename matches "bib*.xml" as a regular expression (e.g., bib90.xml, bib91.xml, ...)
s(`foo.xml`)	The specific document "foo.xml"
s(`*`, `schema.dtd`)	All known XML documents that conform to "schema.dtd"
s([`book*.xml`, `article*.xml`], [`book.dtd`, `article.dtd`]	All XML documents whose filenames match "book*.xml" or "article*.xml" and conform to "book.dtd" or "article.dtd"

Table 1: Different forms of the Source (s) operator

5.2 Follow (ϕ)

The *Follow* operator, denoted by the symbol ϕ, takes as its argument a path expression in entry point notation as defined in Section 4.3. It finds the elements of each bag that correspond to the entry point. For

each such element it extracts the vertices reachable by the path expression. There are two versions of the Follow operator: the *Unnesting follow*, denoted by ϕ_μ and the *Straight follow*, denoted by ϕ_s. The semantics of these two versions are:

Unnesting follow: For each reachable vertex, a new bag is generated consisting of the contents of the original bag augmented with the new vertex. If there is more than one vertex reachable by the same path expression, the unnesting realization of the Follow operator – ϕ_μ – unnests all such vertices augmenting the size of each input bag by one and increasing the number of bags in the collection.

Straight follow: The Straight Follow operator – ϕ_s performs the same operation as the unnesting version except that, in the resulting bag, it does not preserve the bag element it operates on.

There is a second optional parameter, which denotes a variable binding for the vertices reached by the *Follow* operator. This binding is propagated onto the bag tags so it can be used subsequently. Figure 5 illustrates both versions of the Follow operator.

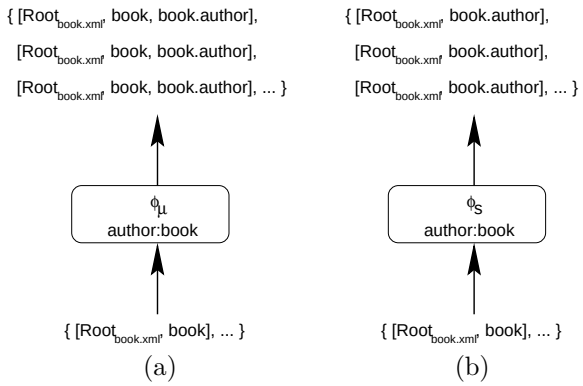


Figure 5: (a) The Unnesting (ϕ_u) and (b) the Straight (ϕ_s) *Follow* operators

5.3 Select (σ)

The basic *Select* operator – σ – filters the bags of a collection using a predicate that can be any combination of logical operators ($\wedge$, $\vee$, $\neg$) and simple qualifications ($=$, $\neq$, $>$, $<$, $\leq$, $\geq$). The predicate can also be any combination of *text-in-context* predicates from the ones in Table 2, which is a powerful feature of the algebra since it can treat content-based retrieval predicates in the same way that it treats logical operators and qualifications. Text-in-context predicates are all position based and not value based. Elements are identified by a *(starting-position, ending position)* pair, while keywords are identified by one position within an XML document. Consequently, it is possible to specify predicates of arbitrary complexity such as: [(`book.author = Hull`) AND (`book.title` contains `Foundations`)]. For a unified framework of content-based query execution over the Internet, this is what glues the text-in-context and traditional query execution elements together.

Operator	Meaning
A $\triangleright$ B	Element A should contain B, regardless of position
A $\triangleleft$ B	B should be contained in element A, regardless of position
A $\dot{\triangleright}$ B	Element A should *directly* contain B
A $\dot{\triangleleft}$ B	B should be *directly* contained in element A
A .. B	The positional distance between A and B (regardless of whether they are elements or content)
A $\oslash$ B	Set difference of A and B
A $\doteq$ B	The content (if it is an element), or the value (if it is a keyword) of A, should be B

Table 2: Text retrieval operators

The Select operator qualifies only the bags of a collection whose tested elements satisfy the predicate. If more than one vertex is tested within a single bag, the bag qualifies if at least one of these vertices satisfies the predicate (*existential semantics*[3]).

5.4 Join ($\bowtie$)

A *Join* operator – $\bowtie$ – joins two collections on a predicate. The predicate can be as generic as the conditions accepted by a selection operator, following the same semantics. The output of a join is the concatenation of pairs of bags that satisfy the predicate. This is depicted in Figure 6.

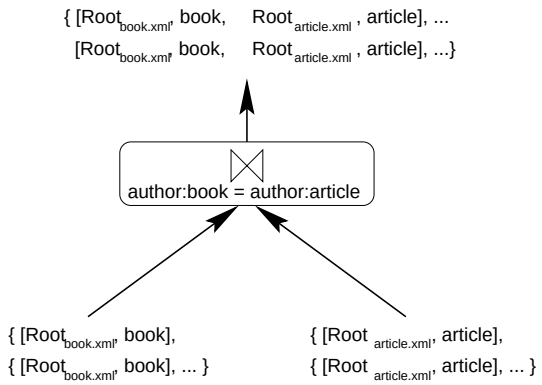


Figure 6: An example of a Join operation

[3]The negated existential semantics are such that NOT (A = B) is equivalent to A $\neq$ B

5.5 Rename (ρ)

The *Rename* operator – ρ – changes the entry point annotation of the elements of a bag. For instance, $\rho(\texttt{book.author}, \texttt{old-author})$ changes the annotation of all bag elements annotated as `book.author` to `old-author`[4] The *Rename* operator is useful in the following cases:

- Eliminating entry points with the same name, in the case of self-joins (i.e., joins between a collection and itself).
- Multiple applications of *Follow* using the same path expression. For instance, the case in which we want to find a book with two authors that share the same last name. To distinguish between them, we can rename one of the path expressions, right after we apply the first *Follow* operator, as is the case in the following expression:

$$\begin{aligned}&\phi_\mu(\texttt{author.lastname:book})[\\ &\quad \rho(\texttt{book.author.lastname, lastname-one})[\\ &\quad \phi_\mu(\texttt{author.lastname:book})[\phi_\mu(\texttt{book})[\ldots]]]\end{aligned}$$

  This allows us to evaluate the following predicate after the second *Follow*:

```
lastname-one = book.author.lastname
```

5.6 Expose (ϵ)

To retrieve specific elements of a bag we need the *Expose* operator – ϵ – which is one of the algebra's construction operators. The *Expose* operator accepts as its parameters a list of path expressions to be exposed from the bags on which it operates, with the path expressions in entry-point notation. The output of an *Expose* operator imposes a new ordering, the same as the order of its arguments; the bag elements no longer have the order they had in the original document. Once the vertices denoted by the path expression are reached, a new *element content* is constructed. The element content can be thought of as all the descendants of a vertex, as shown in Figure 7. For the output to be valid XML, there must be a facility to create actual elements. This is the work of the *Vertex* operator.

5.7 Vertex (v)

The *Vertex* operator – v – creates the actual XML vertex that will encompass everything created by an *Expose* operator. It arranges the element content according to the order its input provides. It creates the XML vertex to which elements can be connected, as well as the named edge that leads to the newly defined vertex. To handle arbitrary complexity in the constructed XML elements, one can push *Vertex* operations[5] around

[4] Our implementation of the algebra does *not* explicitly use the *Rename* operator. It uses a combination of variable bindings and query language specific constructs to automatically generate the correct entry points.

[5] A *Vertex* operation is different than the operator, in the sense that it operates on vertices and not bags. Subsequently, for simplicity reasons, we will use the same symbol for the operator and the operation.

the path expressions exposed by an *Expose* operator. Any vertices exposed can be nested, at exposition time, inside a reachable vertex, which can in turn appear inside an other newly created vertex, leading to arbitrary element nesting. Figure 8 depicts a combination of *Vertex* and *Expose* operations.

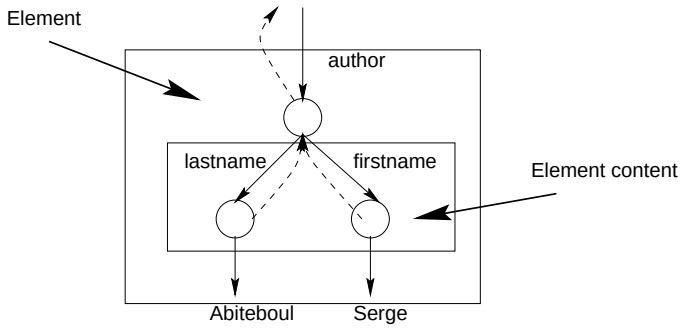


Figure 7: The difference between element contents and elements

The algebra takes care of the distinction between element and element content by providing functors that fetch either the content of the element or the actual element. In most cases, each path expression refers to the content of a vertex[6]. If necessary, one can explicitly state that the path expression refers to the element or the element content. A third functor refers to the *tag* of an element, in other words, the name of the edge that leads to it. Such a facility is necessary in order to provide for operations on edges as well as vertices. Consequently, each path expression can be qualified with either a *content*, an *element* or a *tag* functor. This qualification can also be used by the predicates of selections and joins (e.g., to perform joins on tag names).

$$v(\texttt{newbook})[\epsilon(v(\texttt{newtitle})[\texttt{book.title}], v(\texttt{newauthor})[\texttt{book.author}])]$$

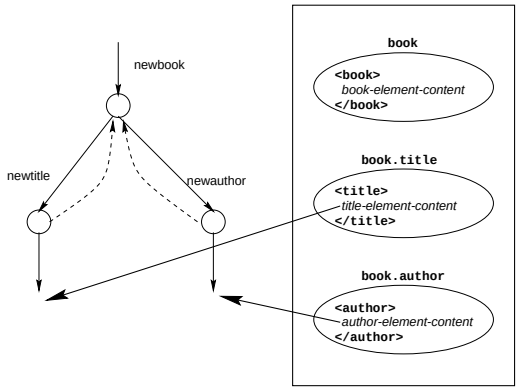


Figure 8: A complex *Vertex* and *Expose* operation and the XML output it generates

[6]This is the assumption we will subsequently use. If no functor is specified, evaluation defaults to the element content.

5.8 Group (γ)

The next algebraic operator is the *Group* – γ – operator. It is used for arbitrary groupings of elements, based on their values. The operator accepts as its parameters a list of path expressions and then proceeds to group the elements of the incoming bags. The order in which the path expressions appear in the parameter list is the grouping order of the resulting bags. In addition, the operator eliminates from the contents of the bag elements that do not appear inside the parameter list. Figure 9 illustrates the operation of the *Group* operator.

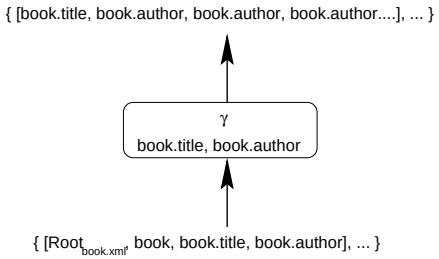


Figure 9: An example of the *Group* operator

Aggregate operations can be easily incorporated into the framework using the logic of the *Group* operator. For instance, to calculate the average price of books by a specific author, one might define an *avg()* functor and use a group operator in the following way:

$$\gamma(\texttt{book.author}, \text{avg}(\texttt{book.price}))$$

This operator will group the bags by the author and for each author compute the average price of her books.

5.9 Union, Intersection, Difference and Cartesian Product

In addition to the XML specific operators defined in the previous sections, the standard set theoretic operators are needed to make the algebra complete.

Union ($\cup$) takes as its input two collections of bags and concatenates them into a single collection. The union is value-based and no duplicates appear in the final output.

Intersection ($\cap$) takes as its input two collections of bags and calculates the value-based intersection of the participating bags.

Difference ($-$) takes as its input two collections of bags and calculates the value-based difference.

Cartesian product ($\times$) is the value-based Cartesian product of two collections of bags. One can think of it a join operation with a predicate that always evaluates to *true*.

The difference between our approach and relational algebra is that the set theoretic operators do not require the bags to be of equal size and have elements of same type (i.e., same entry points). Consequently, the collections of bags which result as the output of such an operation can be heterogeneous.

Table 3 gives a summary of all the algebraic operators that were introduced in the previous sections (Sections 5.1– 5.9).

Operator	**Specification**	**Meaning**
Source	$s(\langle$file list$\rangle$ ($\langle$DTD/XML-Schema list$\rangle$)?)	Specify the source of the expression. Either an XML document, specified by a filename filter or a filename along with a DTD or XML-Schema the document should conform to.
Follow	$\phi(\langle$path$\rangle)$	Follow the specified path inside the elements of a bag. Either unnest (ϕ_μ) augmenting the bags or follow straight (ϕ_s) changing the type of the bag element.
Select	$\sigma_{\text{predicate}}$	Evaluate the *predicate* on incoming bags, filtering out those that do not satisfy it.
Join	$\underset{\text{predicate}}{\bowtie}$	Join two incoming sources on the specified predicate. Filter out the pairs of incoming bags that do not satisfy it.
Rename	$\rho(\langle$from-entry-point$\rangle, \langle$to-entry-point$\rangle)$	Rename the specified entry point.
Expose	$\epsilon(\langle$path$\rangle(, \langle$path$\rangle)\star)$	Expose all the specified path expressions of the incoming bags.
Vertex	$v(\langle$tag$\rangle)$	Create a new vertex and an edge named *tag* that leads to it.
Group	$\gamma(\langle$path$\rangle(, \langle$path$\rangle)\star)$	Group the bags on the specified list of elements, eliminating all elements not appearing in the list.
Union	$\cup$	Value-based union of two collections of bags.
Intersection	$\cap$	Value-based intersection of two collections of bags.
Difference	$-$	Value-based difference of two collections of bags.
Cartesian Product	$\times$	Value-based Cartesian product of two collections of bags.

Table 3: Summary of Logical Operators

6 Miscellaneous characteristics

In this section, we present advanced features of the algebra. We also further justify the choice of using a specialized notation for path expressions and not a standard such as XPath [CD99].

In the introduction to the XML data model (Section 4.1), we defined an ordering on edges and vertices of the XML graph. Thus far, however, we have not used any notion of ordering in the operators, which is because the operators are not sensitive to the ordering of the XML data. Ordering information is captured by the conditions framework through which the filtering operators decide whether to qualify a vertex or not, as well as the path expressions framework that specify the vertices on which the operators act.

To be more precise, there exist ways to define operations between vertices based on their order within the XML graph and not on their content. For instance, the following condition:

$$\text{element}(\langle \text{path}_1 \rangle) =_{\text{order}} \text{element}(\langle \text{path}_2 \rangle)$$

qualifies the elements only if they have the same *ordering* within the document, i.e., if they are the same *physical* XML element, which gives way for more elaborate operations, such as *order joins*. Suppose we have followed an arbitrary number of reference edges to remote vertices. At that point we want to ensure that all the vertices we have followed down to, stem from the same original vertex. Since the algebraic operators keep track of every vertex they have ever visited (except *Straight follow*), by issuing the correct condition on the order of the vertices we can ensure the desired behavior.

An alternative path expression specification mechanism we could have used is XPath [CD99]. The main difference between the two approaches is that we use entry points, while XPaths do not. The only way for XPaths to specify an entry point is through bindings. However, by only supporting bindings at the operator level and not at the path expression level and by explicitly stating the entry point of each path expression we overcome the problem of identifying the starting point of evaluation.

One could argue that if one uses the *parent()/ancestor()* functionality of XPaths, the entry point notation is not needed. Consider a mechanical parts database, in which each part consists of other parts, arbitrarily nested. We want to retrieve the serial number of any part that contains, at any level, parts whose type is "Bolt". The algebraic expression to accomplish that is:

$$\epsilon(\texttt{serialno:part})[\sigma_{\texttt{type:part.*.part = Bolt}}[\phi_{\mu}(\texttt{*.part:part})[\phi_{\mu}(\texttt{part})[s(\texttt{parts.xml})]]]]$$

The *parent()* construct is of no use in this case. If we are operating at the deepest `part` elements and we ask for their parent through `part/parent()`, this will return the immediate parent only, and not the originating one. The *ancestor()* function does not help either; it retrieves *all* the ancestor vertices. Using the entry point notation, the originating parent is captured by specifying the path expression as: `serialno:part`.

Our path expression notation provides the same functionality as the XPath standard in the following ways[7]:

1. XPath supports functions as part of the path expression (e.g., *parent()* to reach the immediate parent of a vertex). For instance, the XPath `lastname/parent()/firstname` (the `firstname` vertex out of the *parent()* of a `lastname` vertex) can be translated into the following path expression in our notation: `firstname:lastname^` (i.e., the `firstname` vertex of `lastname`'s immediate parent.)

2. Many important features of XPath (e.g., transitive closure) can be captured using wild cards as part of our path expressions. For example, the XPath `chapter//title` can be written as `chapter.*.title` in the path expressions our algebra provides. There is also support for recursive path expressions: XPath

[7]Except selections which are handled at the operator level.

`chapter/section*//title` can be written as `chapter.section*.title` in the algebra, yielding the same result.

7 From Queries to Algebraic expressions

This section presents queries expressed in Quilt [CRF00]. The purpose is to show that Quilt queries can be easily translated into algebraic expressions using our algebra. Quilt is a relatively new XML query language that is rapidly gaining in popularity. Its power lies in the fact that it combines features from several languages and application domains. It is designed to be broadly applicable across all types of XML as well as relational data sources.

Consider the following Quilt query, which finds books that have titles but no authors[8]:

```
FOR $b IN //book
WHERE exists($b/title)
AND NOT exists($b/author)
RETURN
<orphan-book>
  $b/title/text()
</orphan-book>
```

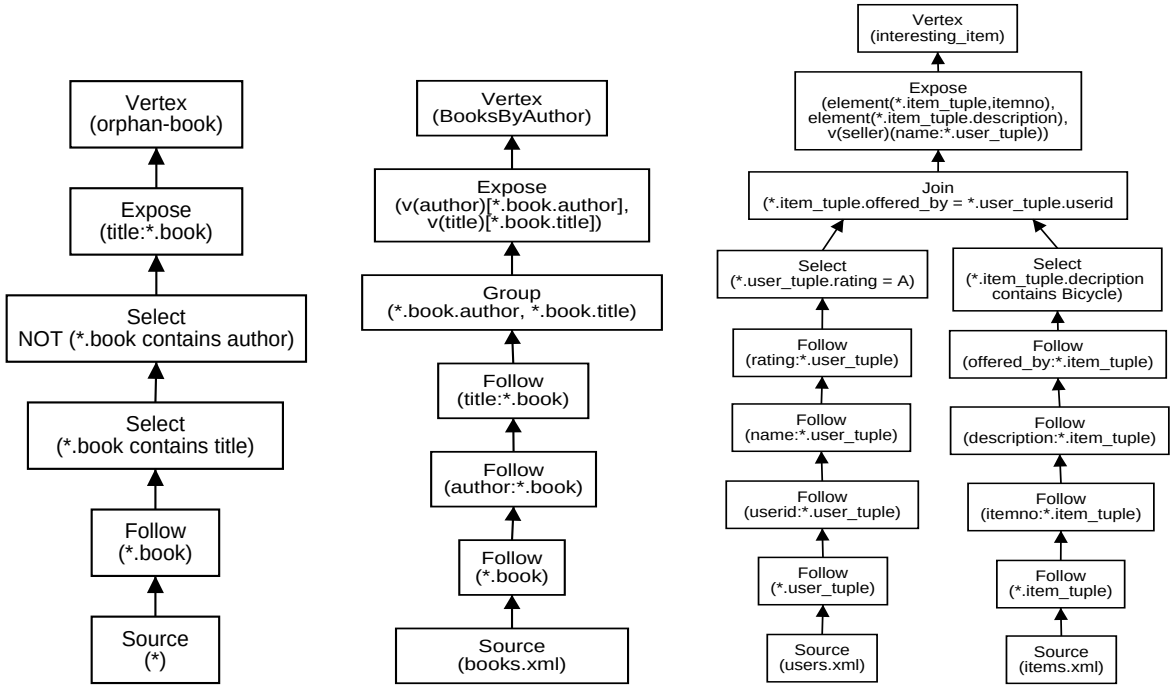
Using the unified predicate framework (i.e., traditional query engine plus text-in-context predicates), the query is easily expressible in the algebra, as Expression 7.1 and Figure 10(a) show.

$$v(\texttt{orphan\_book})[\epsilon(\texttt{title:*.book})[\sigma_{\neg(\texttt{*.book} \triangleright \texttt{author})}[\sigma_{\texttt{*.book} \triangleright \texttt{title}}[\phi_{\mu}(\texttt{*.book})[s(*)]]]]] \tag{7.1}$$

Our algebra is powerful enough to capture grouping conversions of documents. The document of Figure 1(a) groups authors by book titles. Suppose we want to change the grouping to titles by book authors. The following query accomplishes that. The expression for that query is 7.2, depicted in Figure 10(b).

```
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
```

[8] All Quilt examples come from [CRF00].



(a) Text-in-context retrieval (algebraic expression 7.1)

(b) Grouping (algebraic expression 7.2)

(c) Algebraic expression 7.3

Figure 10: Translations of queries into algebraic expressions

```
  )
</BooksByAuthor>
```

$$v(\texttt{BooksByAuthor})[\epsilon(v(\texttt{author})[\texttt{*.book.author}]v(\texttt{title})[\texttt{*.book.title}])[$$
$$\gamma(\texttt{*.book.author}, \texttt{*.book.title})[\phi_\mu(\texttt{title:*.book})[\phi_\mu(\texttt{author:*.book})[$$
$$\phi_\mu(\texttt{*.book})[s(\texttt{books.xml})]]]]]] \tag{7.2}$$

The proposed algebra can also capture other, more advanced Quilt queries. For instance, the following query specifies: "Find item numbers and descriptions of Bicycles offered by users whose rating is "A", and the names of the offering users":

```
<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>
```

The previous query translates into the algebraic expression 7.3, depicted in Figure 10(c).

$$
\begin{aligned}
& v(\texttt{interesting\_item})[\epsilon(\text{element}(\texttt{*.item\_tuple.itemno}),\\
& \quad \text{element}(\texttt{*.item\_tuple.description}), v(\texttt{seller})[\texttt{name:*.user\_tuple}])[\\
& \quad (\sigma_{\texttt{*.item\_tuple.description} \rhd \texttt{Bicycle}}[\phi_\mu(\texttt{offered\_by:*.item\_tuple})[\\
& \quad \phi_\mu(\texttt{description:*.item\_tuple})[\phi_\mu(\texttt{itemno:*.item\_tuple})[\\
& \quad \phi_\mu(\texttt{*.item\_tuple})[s(\texttt{items.xml})]]]]]) \\
& \quad \bowtie_{\texttt{*.item\_tuple.offered\_by = *.user\_tuple.userid}} \\
& \quad (\sigma_{\texttt{*.user\_tuple.rating = A}}[\phi_\mu(\texttt{rating:*.user\_tuple})[\\
& \quad \phi_\mu(\texttt{name:*.user\_tuple})[\phi_\mu(\texttt{userid:*.user\_tuple})[\\
& \quad \phi_\mu(\texttt{*.user\_tuple})[s(\texttt{users.xml})]]]]])]
\end{aligned}
\tag{7.3}
$$

8 Equivalence Rules

The purpose of this section is to present equivalence rules between algebraic expressions, which is the first step toward devising a query optimizer based on this algebra.

8.1 Rule Definitions

The following equivalence rules hold:

$$B\{\phi_\mu(P)[A]\} \equiv B\{\phi_s(P)[A]\}, \quad (8.4)$$
iff B does not operate on P's entry point

$$\phi_\mu(A)[\phi_\mu(B)] \equiv \phi_\mu(B)[\phi_\mu(A)], \quad (8.5)$$
iff $\exists C \prec A, C \prec B : C = A\dot{\sqcap}B,$ or $A \sqcap B = \perp$

$$\sigma_c[A] \equiv \sigma_c\{\phi_s(P)[A]\}, P \vdash c \quad (8.6)$$

$$\sigma_{c_1}[\sigma_{c_2}(A)] \equiv \sigma_{c_2}[\sigma_{c_1}(A)] \quad (8.7)$$

$$\sigma_{c_1}[\sigma_{c_2}(A)] \equiv \sigma_{c_1 \wedge c_2}(A) \quad (8.8)$$

$$\sigma_c\{\phi_\star(P)[A]\} \equiv \phi_\star(P)\{\sigma_c[A]\}, P \nvdash c \quad (8.9)$$

$$\sigma_c[A \underset{d}{\bowtie} B] \equiv A \underset{c \wedge d}{\bowtie} B \quad (8.10)$$

$$\sigma_c[A \bowtie B] \equiv \sigma_{c_1}[A] \bowtie \sigma_{c_2}[B] \quad (8.11)$$

$$A \bowtie B \equiv B \bowtie A \quad (8.12)$$

$$(A \bowtie B) \bowtie C \equiv A \bowtie (B \bowtie C) \quad (8.13)$$

$$\epsilon(P)(B[A]) \equiv B(\epsilon(P)[A]), \quad (8.14)$$
iff B uses only elements exposed by P

$$\epsilon(v(T)[P])(B[A]) \equiv \epsilon(v(T)[P])\{B(\epsilon(P)[A])\}, \quad (8.15)$$
iff B uses only elements exposed by P

$$\epsilon(P_1, P_2)[A \underset{c}{\bowtie} B] \equiv \epsilon(P_1)[A] \underset{c}{\bowtie} \epsilon(P_2)[B], \quad (8.16)$$
iff c uses only elements reachable by P_1 and P_2

$$\epsilon(v(T_1)[P_1], v(T_2)[P_2])[A \underset{c}{\bowtie} B] \equiv \epsilon(v(T_1)[P_1], v(T_2)[P_2])\{\epsilon(P_1)[A] \underset{c}{\bowtie} \epsilon(P_2)[B]\}, \quad (8.17)$$
iff c uses only elements reachable by P_1 and P_2

Table 4 presents the auxiliary operators appearing in the equivalence rules. What follows is an explanation of these rules.

Operator Example	**Meaning**
$A \prec B$	Path expression A is a prefix of B
$A \sqcap B$	The common prefix of path expressions A and B
$A\dot{\sqcap}B$	The greatest common prefix of path expressions A and B
$\perp$	The null path expression
$P \vdash q$	Condition q refers to path expression P

Table 4: Auxiliary operators for the equivalence rules

Rule 8.4 deals with the equivalence of the *unnesting* and the *straight* versions of the *Follow* operator. One can use them interchangeably, as long as ϕ_s's path expression's entry point is not used in the future.

Rule 8.5 is about the interchangeability of the *Follow* operators. It is not always possible to interchange unnesting *Follow* operators; it is only possible if the two operators refer to path expressions that have a common entry point, or they refer to entirely different elements of a bag. Consider the following situation on the XML document of Figure 1(a):

$$\phi_{\mu}(\texttt{lastname:book.author})[\phi_{\mu}(\texttt{author:book})]$$

This operation is not equivalent to:

$$\phi_{\mu}(\texttt{author:book})[\phi_{\mu}(\texttt{lastname:book.author})]$$

since the inner *Follow* operator tries to evaluate a path expression on a non-existing entry point, which is not the case in the second example. Had the original example been:

$$\phi_{\mu}(\texttt{title:book})[\phi_{\mu}(\texttt{author:book})]$$

the interchange of the order of the operators would have been permitted, since both path expressions share a common prefix (`book`). The second case in which interchange is permitted is when the operators refer to entirely different path expressions. For instance:

$$\phi_{\mu}(\texttt{title:book})[\phi_{\mu}(\texttt{title:article})] \equiv \phi_{\mu}(\texttt{title:article})[\phi_{\mu}(\texttt{title:book})]$$

Rule 8.6 states that if a path expression inside a selection condition appears, it is possible for this path expression to be followed prior to the selection application. The *Follow* operation should be a straight one, meaning that the elements of the bag operated on (denoted by the path expression) have to change. The *Follow* operation cannot be an unnesting one, since, in that case, the size of the resulting bags would potentially change.

Rule 8.7 deals with selection associativity. Given two selections on the same expression, it is possible to interchange their order.

Rule 8.8 points out that given two selections on the same expression, they can be collapsed into a single selection using the conjunction of the respective predicates.

Rule 8.9 is about pushing down selections with respect to the *Follow* operator. If the selection is imposed over a *Follow* operator it is possible to interchange the order only if the path expression followed is not part of the selection condition. The selection can be carried out on the proper subset of the bag that does not contain the followed path expression. The symbol $\phi_{\star}$ denotes that this rule is independent of whether there is an unnesting or straight *Follow* operator participating in the expression.

Rule 8.10 deals with the situation in which there is a selection after a join. It is possible to conjoin the selection and the join predicate.

Rule 8.11 shows that if there is a selection following a join, and this condition is a conjoin of conditions, each of which refers to only one of the join inputs, it is possible to distribute the selection over the join inputs, creating two new selections.

Rule 8.12 deals with join commutativity. Given that all operators operate on bags, the output of a join is the concatenation or pairs of joining bags and bags are unordered, the resulting bags of the *Join* operation are insensitive to the order in which the join inputs are specified.

Rule 8.13 is about join associativity. By construction, bags are unordered, so the resulting bag of the whole *Join* operation will be the same regardless of the execution order of the individual *Join* operations.

Rules 8.14 and 8.15 deal with pushing down element expositions. It is only allowed if the eliminated bag elements are not used in subsequent operations. If there are vertex operations over the exposed paths, then the original *Expose* operator must remain in the expression. Otherwise, all subsequent references to the exposed paths would have to be adjusted to include the new vertex names.

Rules 8.16 and 8.17 are about the distribution of expositions across the branches of a join operation. Conditions similar to those in Rules 8.14 and 8.15 should be satisfied, with the only difference being that the join predicate contains only exposed path expressions.

8.2 Rule Applications

In this section we present various applications of the aforementioned equivalence rules in the context of a prototype system that implements the proposed algebra for the logical representation of query plans. By applying the equivalence rules, we can optimize query processing time through:

Follow ordering (Rule 8.5) allows the system to optimize following based upon the selectivity of path expressions. For instance, suppose we are operating on `book` elements and we wish to follow down to `author` and `editor` elements. Assuming that `author` elements are always present inside a `book` element, while `editor` elements may be not, it is more efficient during query processing to apply the `editor` element retrieval first. This will limit the input size to the subsequent *Follow* operator since it will eliminate all `book`s that do not have `editor`s.

Join commutativity and associativity (Rules 8.12 and 8.13) allows us to treat join ordering using existing optimization techniques. To our knowledge, no other algebraic framework for pure XML data treats join operations in that manner (see Section 4.2).

Selection distribution and interchangeability (Rules 8.7, 8.9 and 8.11) are useful for pushing down selections to decrease the size of the input to subsequent operations. In our system, sequences of *Follow* and *Select* operators can be used to generate a single search engine retrieval predicate. Since both path expressions and predicates have varying selectivities, it is important to be able to push down the more selective operators.

Elimination of unused bag elements (Rules 8.4, 8.16 and 8.17) is useful in that it allows us to dispose of unused parts of documents during query processing and to limit the operation only to the parts that will be used in later stages of the query.

9 Conclusions and Further Work

We have presented a framework for representing queries over XML data in an algebraic way. The operators we have defined both explore *and* filter XML data as well as constructing new XML data. The contribution of this work is that it introduces an algebra that operates on a new data model. Because our operators work on collections of bags of vertices, we are able to produce far simpler expressions than other, previously proposed approaches. Our framework can also be used in conjunction with traditional cost-based query optimization techniques. Moreover, it provides room for further optimization specific to the XML data model (e.g., collapsing combinations of *Select/Follow* operations into a single access path).

In the future, we plan to further evaluate our algebra in the context of our Internet query engine project. We also want to extend the algebra to support some of the more advanced features of XML query languages, such as the *filter* operator of Quilt. Currently, a large subset of this algebra is being used by our query engine to formulate logical execution plans, which are then directly translated into physical plans using some ad-hoc heuristics. To prove the utility of using an algebraic logical plan representation, we are in the process of writing an optimizer for our algebra. We hope to demonstrate through this process that it is indeed possible to optimize queries over unstructured XML data.

References

[ABC+00] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible stylesheet language (XSL). http://www.w3.org/TR/xsl/, November 2000. W3C Candidate Recommendation.

[Abi97] Serge Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 1997.

[BMG93] José A. Blakeley, William J. McKenna, and Goetz Graefe. Experiences building the open OODB query optimizer. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 287–296. ACM Press, 1993.

[BMR99a] David Beech, Ashok Malhotra, and Michael Rys. A formal data model and algebra for XML, September 1999. W3C XML Query working group note.

[BMR99b] David Beech, Ashok 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.

[BPCSM98] T. Bray, J. Paoli, and C-Sperberg-McQueen. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, February 1998. W3C Recommendation.

[Bun97] P. Buneman. Semi-structured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tuscon, AZ, May 1997.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD Conference*, Minneapolis, MN, May 1994.

[CBB+00] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *Object Data Standard ODMG 3.0*. Morgan-Kaufman, January 2000.

[CCB94] Charles 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.

[CD99] James
Clark and Steve DeRose. XML path language (XPath), version 1.0. http://www.w3.org/TR/xpath, November 1999. W3C Recommendation.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagara-CQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, May 2000*, pages 379–390, 2000.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod72] E. F. Codd. Relational completeness of database sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, Dallas, TX, May 2000.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML, August 1998. Submission to W3C available at http://www.w3.org/TR/NOTE-xml-ql.

[FM95a] Leonidas Fegaras and David Maier. An algebraic framework for physical OODB design. In Paolo Atzeni and Val Tannen, editors, *Database Programming Languages (DBPL-5), Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995*, Electronic Workshops in Computing. Springer, 1995.

[FM95b] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 47–58. ACM Press, 1995.

[FSSW] Mary Fernandez, Jerome Simeon, Dan Suciu, and Philip Wadler. A data model and algebra for XML query. http://www.cs.bell-labs.com/wadler/topics/xml.html#algebra.

[MAG+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] David 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. In *Proceedings of the 25th VLDB Conference*, pages 315–326. Morgan-Kaufmann, September 1999.

[NDea] Jeffrey Naughton, David DeWitt, and David Maier et. al. The Niagara internet query system. Submitted for Publication.

[STD+00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a network query engine for producing partial results. In *WebDB 2000*, Dallas, TX, May 2000.

[W3C] World wide web consortium. http://www.w3c.org.