The Importance of Algebra for XML Query Processing

Stelios Paparizos and H. V. Jagadish

University of Michigan, Ann Arbor, MI, USA
{spapariz, jag}@umich.edu
Supported in part by NSF, under grant IIS-0208852

Abstract. Relational algebra has been a crucial foundation for relational database systems, and has played a large role in enabling their success. A corresponding XML algebra for XML query processing has been more elusive, due to the comparative complexity of XML, and its history. We argue that having a sound algebraic basis remains important nonetheless. In this paper, we show how the complexity of XML can be modeled effectively in a simple algebra, and how the conceptual clarity attained thereby can lead to significant benefits.

1 Introduction

XML is in wide use today, in large part on account of its flexibility in allowing repeated and missing sub-elements. However, this flexibility makes it challenging to develop an XML data management system.

The first XML enabled systems used a native navigational model to query documents. They showed a lot of promise and the interest shifted into them. In such model, an XML document is first loaded into memory as one big tree and then path expressions are matched via traversals in this tree. Examples of systems using this model include [**?**,**?**,**?**]. A big limitation of such systems is that the biggest XML document they can process is limited by the amount of physical memory in the system. Since they are essentially instance-at-a-time systems, they have little room for optimization and indices. In general they are known to provide full support of XQuery but poor performance – a traversal for a descendant node can force a search of the entire database.

Another popular approach (perhaps, due to existing system support) was to use relational databases and map XML documents into relations. Such solutions were presented in [**?**,**?**,**?**,**?**,**?**]. But these mappings may come at the expense of efficiency and performance since relational databases were not designed for XML data. The data transformation process and the query time can be lengthy. Many expensive operations (e.g. joins) needed to determine structural relationships between elements. Due to the variations possible in parallel elements within XML, and the possibility of repeated and missing sub-elements, the mapping becomes non-trivial, and frequently requires liberal use of null values or of un-normalized relations. Overall, with relational support for XML expressiveness is not the issue - with sufficient effort, many mappings are possible. The issue is how natural the mapping is, how well it preserves the original structure and how expensive the resulting relational expressions can be to evaluate.

The solution to this problem would be to use a native XML system while taking advantage of the knowledge of many years of research in the relational world. For example, one of the lessons we learned is that for efficiency purposes, it is essential to use

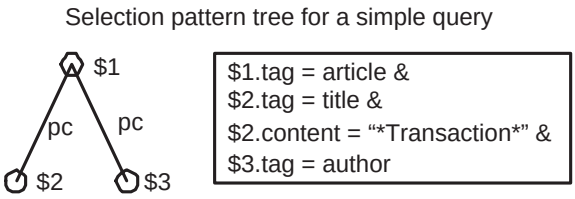

Fig. 1. A pattern tree corresponding to the query ‘Select articles with some author and with title that contains Transaction’.

‘set-at-a-time’ processing instead of ‘instance-at-a-time’. We also know that the careful choice of evaluation plan can make orders of magnitude difference in query cost. And finally, the relational algebra is long recognized as a useful intermediate form in query processing. From all of the above we conclude that a set-oriented bulk algebra is essential in XML query processing.

Yet, although native algebraic solutions appear to be the better system approach (with significant existing efforts like those in [**?**,**?**]), there is no universally accepted XML algebra. In this paper we present our algebraic XML solution and demonstrate its strengths. We start by introducing our basis, the pattern trees and show a Tree Algebra for XML in Section 2. We continue with a discussion in XML and XQuery order related issues in Section 3. Then we show some practical advances in XML algebras in Section 4 and some optimization opportunities that arise in Section 5. We conclude the paper with a few final words in Section 6.

2 Ideal Algebra - Tree Algebra for XML

In the relational model, a tuple is the basic unit of operation and a relation is a set of tuples. In XML, a database is often described as a forest of rooted node-labeled trees. Hence, for the basic unit and central construct of our algebra, we choose an *XML query pattern* (or *twig*), which is represented as a rooted node-labeled tree. An example of such tree, we call it *pattern tree*, is shown in Figure 1. An edge in such tree represents a structural inclusion relationship, between the elements represented by the respective pattern tree nodes. The inclusion relationship can be specified to be either immediate (parent-child relationship) or of arbitrary depth (ancestor-descendant relationship). Nodes in the pattern tree usually have associated conditions on tag names or content values.

Given an XML database and a query pattern, the *witness trees* (pattern tree matchings) of the query pattern against the database are a forest such that each witness tree consists of a vector of data nodes from the database, each matches to one pattern tree node in the query pattern, and the relationships between the nodes in the database satisfy the desired structural relationship specified by the edges in the query pattern. The set of witness trees obtained from a pattern tree match are all structurally identical. Thus, a pattern tree match against a variegated input can be used to generate a structurally homogeneous input to an algebraic operator. Sample of witness trees can be found in Figure 2.

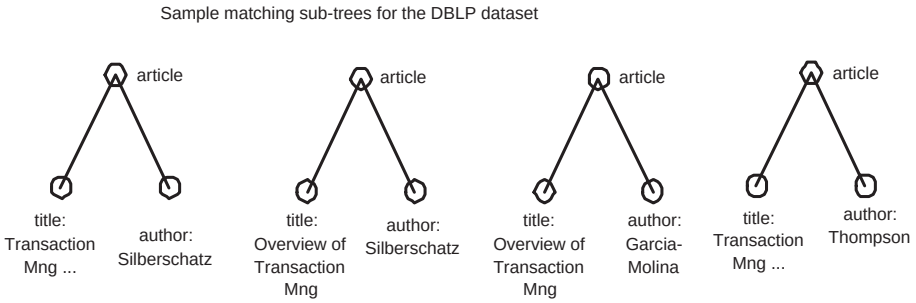

Fig. 2. A sample of the resulting witness trees produced from the matching of the tree in Figure 1 to the DBLP dataset.

1. ORDER BY clause,
   explicit, depends on value.
2. Re-establish original document order,
   implicit, required by XML .
3. Binding order of variables,
   implicit, depends on variable binding predicates.

Fig. 3. Ordering Requirements for XML and XQuery

Using this basic primitives, we developed an algebra, called *Tree Algebra for XML* (TAX), for manipulating XML data modeled as forests of labeled ordered trees. Motivated both by aesthetic considerations of intuitiveness, and by efficient computability and amenability to optimization, we developed TAX as a natural extension of relational algebra, with a small set of operators. TAX is complete for relational algebra extended with aggregation, and can express most queries expressible in popular XML query languages. Details about the algebra, with illustrative examples can be found in [**?**].

3 Ordering and Duplicates

XML itself incorporates semantics in the order data is specified. XML queries have to respect that and produce results based on the order of the original document. XQuery takes this concept even further and adds an extra implicit ordering requirement. The order of the generated output is sensitive to the order the variable binding occurred in the query, we call this notion *'binding order'*. Additionally, a FLWOR statement in XQuery may include an explicit *ORDERBY* clause, specifying the ordering of the output based on the value of some expression – this is similar in concept with ordering in the relational world and SQL. To facilitate our discussion we summarize the XML and XQuery order properties in Figure 3.

Although XML and XQuery require ordering, many "database-style" applications could not care less about order. This leaves the query processing engine designer in a quandary: should order be maintained, as required by the semantics, irrespective of the additional cost; or can order be ignored for performance reasons. What we would

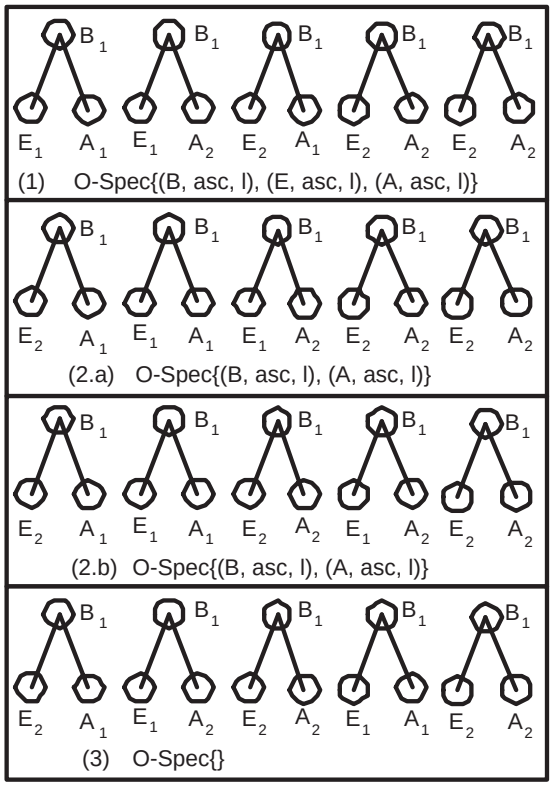

Fig. 4. Collections with Ordering Specification O-Spec. A "fully-ordered" collection in (1), two variations of a "partially-ordered" collection in (2.a) and (2.b) and an unordered collection in (3). Duplicates are allowed in these examples, so (3) is not a set.

like is an engine where we pay the cost to maintain order when we need it, and do not incur this overhead when it is not necessary. In algebraic terms, the question we ask is whether we are manipulating sets, which do not establish order among their elements, or manipulating sequences, which do.

The solution we propose is to define a new generic *Hybrid Collection* type, which could be a set or a sequence or even something else. We associate with each collection an *Ordering Specification O-Spec* that indicates precisely what type of order, if any, is to be maintained in this collection.

As an example, in Figure 4, we can see a few ordered collections using the *Ordering Specification*. A simple sorting procedure using all B nodes (identifiers), sorting them in ascending order and placing all empty entries in the beginning is described by (B, asc, l). In part (1) of the Figure we see a "fully-ordered" collection; all the nodes in every tree were used to perform the sort. A "fully-ordered" collection has one and only one way that the trees can be ordered (absolute order). In parts (2.a) and (2.b) we see the same "partially-ordered" collection; only nodes in parts of every tree were used to

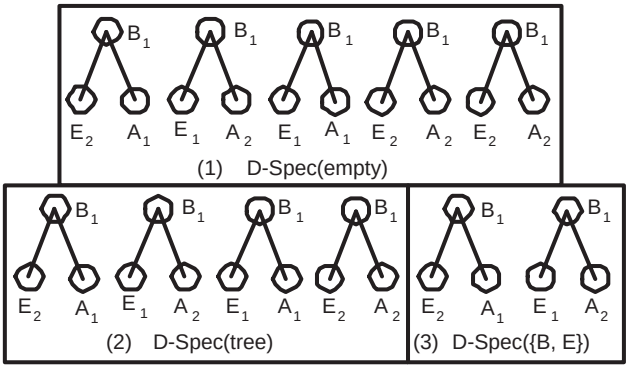

Fig. 5. Collections of trees coupled with their Duplicate Specification D-Spec. All types of duplicates are allowed in (1), deep-tree comparison is used to eliminate duplicates in (2) and duplicates are eliminated in (3) using a partial comparison on B and E nodes. Both (2) and (3) can be thought as sets since no duplicates exist.

perform the sort. A "partially-ordered" collection can potentially have multiple ways it can be ordered. Parts (2.a) and (2.b) show the same collection ordered by the same key with clearly more than one representations of the absolute tree order. In part (3) we see a collection with unspecified order (any order).

Duplicates in collections are also a topic of interest, not just for XML, but for relational data as well. In relational query processing, duplicate removal is generally considered expensive, and avoided where possible even though relational algebra formally manipulates sets that do not admit duplicates.The more complex structure of XML data raises more questions of what is equality and what is a duplicate. Therefore there is room for more options than just sets and multi-sets. Our solution is to extend the *Hybrid Collection* type with an explicit *Duplicate Specification D-Spec* as well.

An example of collections with *D-Spec* describing how duplicates were previously removed from them can be found in Figure 5. Notice the duplicates that exist in part (1), how the last tree from (1) is removed in part (2) and how multiple trees are removed in part (3).

Using our Hybrid Collections we extended our algebra. Thus, we were able to develop query plans that maintain as little order as possible during query execution, while producing the correct query results and managing to optimize duplicate elimination steps. Formal definitions of our collections, extensions to algebraic operations along with the algorithm for correct order can be found in [**?**]. We believe our approach can be adopted in any XML algebra, providing correctness and flexibility.

4 Tree Logical Classes (TLC) for XML

XQuery semantics frequently requires that nodes be clustered based on the presence of specified structural relationships. For example the RETURN clause requires the complete subtree rooted at each qualifying node. A traditional pattern tree match returns a set of 'flat' witness trees satisfying the pattern, thus requiring a succeeding grouping

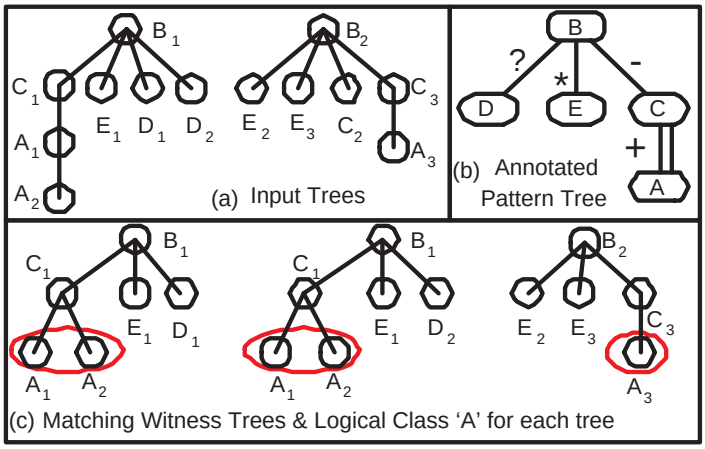

Fig. 6. A sample match for an Annotated Pattern Tree.

step on the parent (or root) node. Additionally, in tree algebras, each algebraic operator typically performs its own pattern tree match, redoing the same selection time and time again. Intermediate results may lose track of previous pattern matching information and can no longer identify data nodes that match to a specific pattern tree node in an earlier operator. This redundant work is unavoidable for operators that require a homogeneous set as their input without the means for that procedure to persist.

The loss of *Structural Clustering*, the *Redundant Accesses* and the *Redundant Tree Matching* procedures are problems caused due to the witness trees having to be similar to the input pattern tree, i.e. have the same size and structure. This requirement resulted in homogeneous witness trees in an inherently heterogeneous XML world with missing and repeated sub-elements, thus requiring extra work to reconstruct the appropriate structure when needed in a query plan. Our solution uses *Annotated Pattern Trees (APTs)* and *Logical Classes (LCs)* to overcome that limitation.

Annotated Pattern Trees accept edge matching specifications that can lift the restriction of the traditional one-to-one relationship between pattern tree node and witness tree node. These specifications can be "-" (exactly one), "?" (zero or one),"+" (one or more) and "*" (zero or more). Figure 6 shows the example match for an annotated pattern tree. The figure illustrates how annotated pattern trees address heterogeneity on both dimensions (height and width) using variations of annotated edges. So A_1, A_2 and E_2, E_3 are matched into clustered siblings due to the "+" and "*" edges in the APT. On the flip side D_1, D_2 matchings will produce two witness trees for the first input tree (the second tree is let through, although there is no D matching) due to the "?" edge in the APT.

Once the pattern tree match has occurred we must have a logical method to access the matched nodes without having to reapply a pattern tree matching or navigate to them. For example, if we would like to evaluate a predicate on (some attribute of) the "A" node in Figure 6, how can we say precisely which node we mean? The solution to his problem is provided by our Logical Classes. Basically, each node in an annotated

```
FOR $a IN distinct-values(document("bib.xml")//author)
RETURN
    <authorpubs>
        { $a }
        {FOR $b IN document("bib.xml")//article
            WHERE $a = $b/author
            RETURN $b/title}
    </authorpubs>
```

Fig. 7. Query 1: Group by *author* query (After XQuery use case 1.1.9.4 Q4).

pattern tree is mapped to a set of matching nodes in *each* resulting witness tree – such set of nodes is called a *Logical Class*. For example in Figure 6, the red(gray) circle indicates how the A nodes form a logical class for each witness tree. Every node in every tree in any intermediate result is marked as member of at least one logical class[1]. We also permit predicates on logical class membership as part of an annotated pattern tree specification, thus allowing operators late in the plan to reuse pattern tree matches computed earlier.

Using this techniques we extended TAX into our *Tree Logical Class* (TLC) algebra. We discuss our operators and our translation algorithm for XQuery to TLC along with other details in [**?**]. We base our discussion on our physical algebra seen in [**?**].

5 Algebraic Optimizations

In this section we demonstrate some of the advantages we get by using algebraic primitives to produce more efficient solutions. We discuss how we address grouping in XQuery and also show some algebraic rewrites that focus on smart placement of ordering and duplicate operations.

Grouping: While SQL allows for grouping operations to be specified explicitly, XQuery provides only implicit methods to write such queries. For example consider a query that seeks to output, for each `author`, `titles` of `articles` he or she is an `author` of (in a bibliography database). A possible XQuery statement for this query is shown in Figure 7. A direct implementation of this query as written would involve two distinct retrievals from the bibliography database, one for `authors` and one for `articles`, followed by a join. Yet, one of our basic primitives of our algebra is a GROUPBY operator, thus enabling us to produce a 'smarter' plan than the one dictated by XQuery.

The procedure uses a rewrite that operates in the following way. In the beginning, the parser will 'naïvely' try to interpret the query from Figure 7 as a join following the logic specified in XQuery. Then a rewrite will transform the plan into a more efficient one using the GROUPBY operator. First, a selection will be applied on the database using the pattern tree of Figure 8.a followed by a projection. This will produce a collection of trees containing all `article` elements and their `author` and `title` children.

[1] Base data, read directly from the database, has no such association.

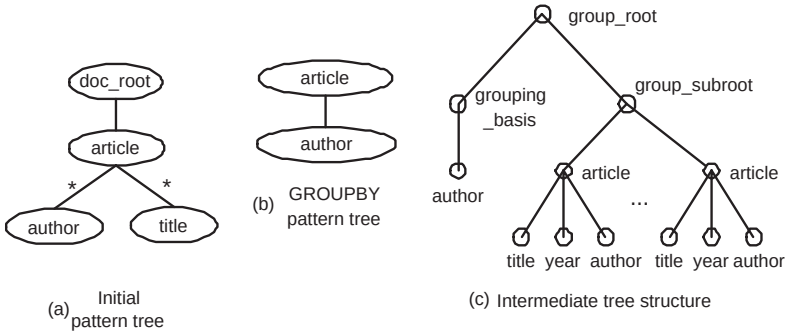

Fig. 8. GROUPBY rewrite for Query 1.

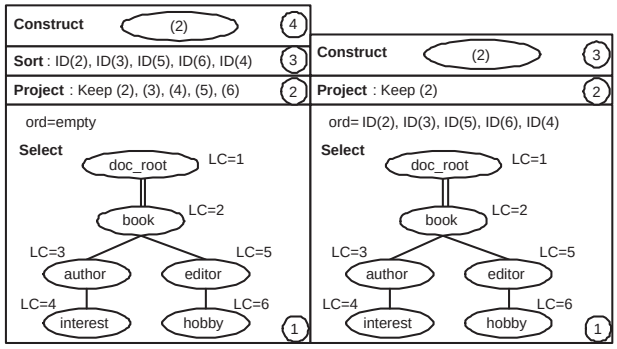

Fig. 9. On the right the rewritten plan having pushed the *Sort* into the *Select.*

Next the input pattern tree to be used by the GROUPBY operator will be generated. For Query 1 this is shown at Figure 8.b. The GROUPBY operator (grouping basis : *author*) will be applied on the generated collection of trees and the intermediate tree structures in Figure 8.c are produced. Finally a projection is done keeping only the necessary nodes.

The power of the algebra allows for the transformation of the 'naïve' join plan into a more efficient query plan using grouping – overcoming the XQuery nuances and making it similar to a relational query asking for the same information. Details of the rewrite algorithm along with more complex scenarios can be found in [**?**].

Duplicates and Ordering: As we discussed in Section 3, smart operation placement of ordering and duplicate elimination procedures can cause orders of magnitude difference in evaluation performance. We show two examples of such rewrites. Details for both techniques along with other examples are found in [**?**].

We start by showing an example on how we optimize ordering in Figure 9. The rewrite takes advantage of our extended operations that use *Ordering Specification* annotations to push the `Sort` procedure into the original `Select`. Thus, the rewrite provides the cost based optimizer with the means to efficiently plan the pattern tree match

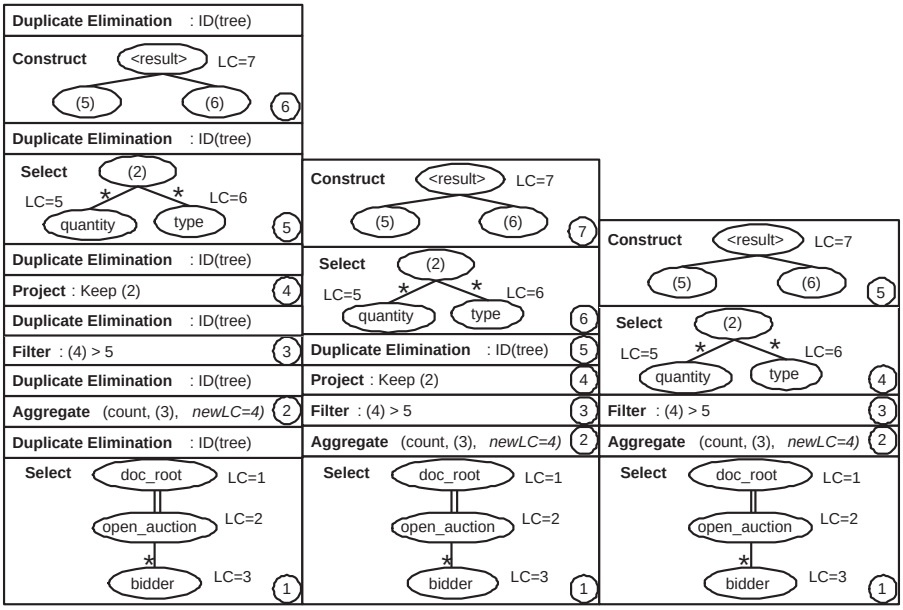

Fig. 10. Minimizing Duplicate Elimination procedures.

using the appropriate physical access methods, without having to satisfy a blocking `Sort` operation at the final step of the query plan.

Additionally, we show an example on how duplicate elimination procedures can be minimized in Figure 10. First, we 'naïvely' force a duplicate elimination after every operation to produce the correct behavior. Then our technique detects and removes all redundant procedures by checking which operations will potentially produce duplicates. With the last step, we take advantage of our 'partial' duplicate collections and manage to remove the duplicate elimination procedure completely.

6 Final Words

The flexibility of XML poses a significant challenge to query processing: it is hard to perform set-oriented bulk operations on heterogeneous sets. In this paper, we proposed our algebraic framework as an effective means to address this problem. We introduced the *Pattern Tree* as the basic unit of operations and developed an algebra that consumes and produces collections of trees in TAX. We discussed how we address efficiently the very complex ordering requirements of XML and XQuery using our *Hybrid Collections*. We showed some practical extensions to tree algebras with the *Annotated Pattern Trees* and *Logical Classes* of TLC. Finally, we hinted on some optimization opportunities that arise with the use of algebra.

It is our belief that a good algebra is central to a good database system, whether relational or XML. The algebraic infrastructure we described in this paper is the basis for the TIMBER [**?**,**?**,**?**] native XML database system developed at the University of

Michigan. The experience we gained from implementing our system strengthens our belief in an algebraic approach. We hope in the near future the XML community can converge on an algebra that combines the best characteristics of the various proposed solutions and provides an alternative formal representation of XQuery.