Grouping in XML

Stelios Paparizos[1], Shurug Al-Khalifa[1], H. V. Jagadish[1], Laks Lakshmanan[2], Andrew Nierman[1], Divesh Srivastava[3], and Yuqing Wu[1]

[1] University of Michigan, Ann Arbor, MI, USA
{spapariz, shurug, jag, andrewdn, yuwu}@umich.edu
Supported in part by NSF, under grants IIS-9986030, DMI-0075447, and IIS-0208852
[2] University of British Columbia, Vancouver, BC, Canada
laks@cs.ubc.ca
[3] AT&T Labs Research, Florham Park, NJ, USA
divesh@research.att.com

Abstract. XML permits repeated and missing sub-elements, and missing attributes. We discuss the consequent implications on grouping, both with respect to specification and with respect to implementation. The techniques described here have been implemented in the TIMBER native XML database system being developed at the University of Michigan.

1 Introduction

Consider a bibliographic database, such as the well-known DBLP repository [5]. Articles have authors, but the number of authors is not the same for each article. Some articles have one author, others have two, three, or more. Yet other articles may have no authors at all. A major strength of XML is that this sort of variation in the data is expressed effortlessly.

Now consider a simple query that seeks to output, for each DBLP author, titles of articles he or she is an author of (in our bibliography database). A possible XQuery statement for this query is shown below. (In fact, this query is a small variation on use case number 1.1.9.4 Q4 in the XQuery specification [4])

```
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>
```

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

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. In XML, given that links are already in place between each article and its authors, one expects that a more efficient implementation might be possible.

The rich structure of XML allows complex grouping specification. For example, we could modify the above query to group not by author but by author's institution. This results in a modified query as follows:

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

The trend initiated by the above query can be extended further, with arbitrary expressions used for grouping. For instance, we may be interested in grouping by both author and institution, as follows:

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

In short, queries that appear to have "grouping" in them are expressed in XQuery without explicit use of a grouping construct. Introducing such a construct appears to be non-trivial on account of the richness and heterogeneity of XML. Yet, explicitly recognizing the grouping operation can lead to more efficient query evaluation. In this paper, we study the issues involved in the use of

grouping in XML query, and the benefits to be derived therefrom. We do it in the context of the TIMBER[23] native XML database system being implemented at the University of Michigan, and the TAX algebra on which it is based.

We discuss how to specify grouping in Sec 3 after a brief introduction to TAX in Sec. 2. We show how to use the grouping operator in a variety of contexts in Sec. 4. In particular, we demonstrate powerful algebraic rewriting rules that can result in the unnesting of XQuery expressions, and the efficient evaluation of queries with grouping. We turn to implementation concerns in Sec. 5 and present experimental results in Sec. 6. A discussion of related work in Sec. 7 is followed by conclusions in Sec. 8.

2 Tree Algebra

An XML document is a tree, with each edge in the tree representing element nesting (or containment). XML also permits references, which are represented as non-tree edges, and may be used in some queries. These are important to handle, and our algebra is able to express these. However, there is a qualitative difference between these reference edges, which are handled as "joins", and containment edges, which are handled as part of a "selection".

To be able to obtain efficient processing on large databases, we require set-at-a-time processing of data. In other words, we require a bulk algebra that can manipulate sets of trees: each operator on this algebra would take one or more sets of trees as input and produce a set of trees as output. Using relational algebra as a guide, we can attempt to develop a suite of operators suited to manipulating trees instead of tuples. We have devised such an algebra, called TAX. Details can be found in [8].

The biggest challenge in devising this algebra is the heterogeneity allowed by XML and XQuery. Each tuple in a relation has identical structure – given a set of tuples from some relation in relational algebra, we can reference components of each tuple unambiguously by attribute name or position. Trees have a more complex structure than tuples. More importantly, sub-elements can often be missing or repeated in XML. As such, it is not possible to reference components of a tree by position or even name. For example, in a bibliographic XML tree, consider a particular book sub-tree, with nested (multiple) author sub-elements. We should be able to impose a predicate of our choice on the first author, on every author, on some (at least one) author, and so on. Each of these possibilities could be required in some application, and these choices are not equivalent.

We solve this problem through the use of *pattern trees* to specify homogeneous tuples of node bindings. For example, a query that looks for articles that have an (at least one) author and a title containing the word "Transaction" is expressed by a pattern tree shown in Figure 1. Matching the pattern tree to the DBLP database, the result is a set of sub-trees rooted at article, each with author and title. A small sample is shown in Figure 2. Such a returned structure, we call a *witness tree*, since it bears witness to the success of the pattern match on the input tree of interest. The set of witness trees produced through the matching

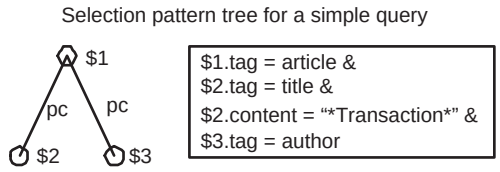

Fig. 1. Pattern Tree for a Query

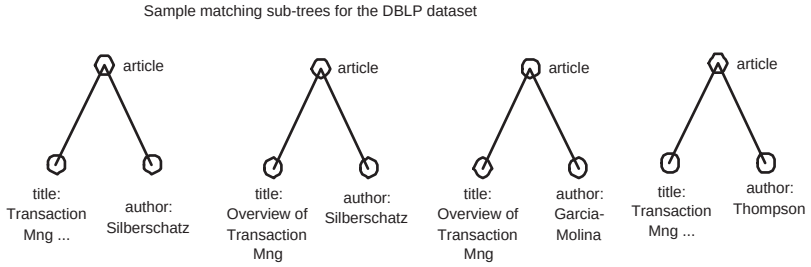

Fig. 2. Witness Trees that Result from a Pattern Match

of a pattern tree are all homogeneous: we can name nodes in the pattern trees, and use these names to refer to the bound nodes in the input data set for each witness tree. A vital property of this technique is that the pattern tree specifies exactly the portion of structure that is of interest in a particular context – all variations of structure irrelevant to the query at hand are rendered immaterial. In short, one can operate on heterogeneous sets of data as if they were completely homogeneous, as long as the places where the elements of the set differ are immaterial to the operation.

The crucial variable-binding FOR clause (and also the LET clause) of XQuery uses a notation almost identical to XPath, which by itself is also used sometimes to query XML data. The key difference between a pattern tree and an XPath expressions is that one XPath expression binds exactly one variable, whereas a single pattern tree can bind as many variables as there are nodes in the pattern tree. As such, when an XQuery expression is translated into the tree algebra, the entire sequence of multiple FOR clauses can frequently be folded into a single pattern tree expression.

All operators in TAX take collections of data trees as input, and produce a collection of data trees as output. TAX is thus a "proper" algebra, with composability and closure. The notion of pattern tree play a pivotal role in many of the operators. Below we give a sample of TAX operators by describe briefly a couple of them, selection and projection. Further details and additional operators can be found in [8].

Selection: The obvious analog in TAX for relational selection is for selection applied to a collection of trees to return the input trees that satisfy a specified selection predicate (specified via a pattern). However, this in itself may not

preserve all the information of interest. Since individual trees can be large, we may be interested not just in knowing that some tree satisfied a given selection predicate, but also the manner of such satisfaction: the "how" in addition to the "what". In other words, we may wish to return the relevant witness tree(s) rather than just a single bit with each data tree in the input to the selection operator.

Selection in TAX takes a collection $\mathcal{C}$ as input, and a pattern $\mathcal{P}$ and adornment SL as parameters, and returns an output collection. Each data tree in the output is the witness tree induced by some embedding of $\mathcal{P}$ into $\mathcal{C}$, modified as possibly prescribed in SL. The adornment list, SL, lists nodes from $\mathcal{P}$ for which not just the nodes themselves, but all descendants, are to be returned in the output. If this adornment list is empty, then just the witness trees are returned. Contents of all nodes are preserved from the input. Also, the relative order among nodes in the input is preserved in the output. Because a specified pattern can match many times in a single tree, selection in TAX is a one-many operation. This notion of selection is strictly more general than relational selection.

Projection: For trees, projection may be regarded as eliminating nodes other than those specified. In the substructure resulting from node elimination, we would expect the (partial) hierarchical relationships between surviving nodes that existed in the input collection to be preserved.

Projection in TAX takes a collection $\mathcal{C}$ as input and a pattern tree $\mathcal{P}$ and a projection list PL as parameters. A projection list is a list of node labels appearing in the pattern $\mathcal{P}$, possibly adorned with $*$. All nodes in the projection list will be returned. A node labeled with a $*$ means that all its descendants will be included in the output. Contents of all nodes are preserved from the input. The relative order among nodes is preserved in the output.

A single input tree could contribute to zero, one, or more output trees in a projection. This number could be zero, if there is no witness to the specified pattern in the given input tree. It could be more than one, if some of the nodes retained from the witnesses to the specified pattern do not have any ancestor-descendant relationships. This notion of projection is strictly more general than relational projection. If we wish to ensure that projection results in no more than one output tree for each input tree, all we have to do is to include the pattern tree's root node in the projection list and add a constraint predicate that the pattern tree's root must be matched only to data tree roots.

In relational algebra, one is dealing with "rectangular" tables, so that selection and projection are orthogonal operations: one chooses rows, the other chooses columns. With trees, we do not have the same "rectangular" structure to our data. As such selection and projection are not so obviously orthogonal. Yet, they are very different and independent operations, and are generalizations of their respective relational counterparts.

3 The Specification of Grouping

In relational databases, tuples in a relation are often grouped together by partitioning the relation on selected attributes – each tuple in a group has the same values for the specified grouping attributes. A source of potential difficulty in trees, is that grouping may not induce a partitioning due to repeated sub-elements. If a book has multiple authors, then grouping books by author will result in this book being repeated as a member of multiple groups.

A deeper point to make is that grouping and aggregation are not separable in relational database systems. The reason is that these operators cause a "type violation": a grouping operator maps a set of tuples to a set of sets of tuples, and an aggregation operator does the inverse. The flexibility of XML permits grouping and aggregation to be included within the formal tree algebra, at the logical level, as distinct operators. In fact, we will see that grouping has a natural direct role to play for restructuring data trees, orthogonally to aggregation.

The objective is to split a collection into subsets of (not necessarily disjoint) data trees and represent each subset as an ordered tree in some meaningful way. As a motivating example, consider a collection of article elements each including its title, authors and so on. We may wish to group this collection by author, thus generating subsets of article elements authored by a given author. Multiple authorship naturally leads to overlapping subsets.

We can represent each subset in any desired manner, e.g., by the alphabetical order of the titles or by the year of publication, and so forth. There is no (value-based) aggregation involved in this task, which involves splitting the collection into subsets and ordering trees within a subset in a specified way. We formalize this as follows.

The groupby operator γ takes a collection as input and the following parameters.

- A pattern tree $\mathcal{P}$; this is the pattern used for grouping. Corresponding to each witness tree T_j of $\mathcal{P}$, we keep track of the source tree I_j from which it was obtained.
- A *grouping basis* that lists elements (by label in $\mathcal{P}$), and/or attributes of elements, whose values are used to partition the set $\mathcal{W}$ of witness trees of $\mathcal{P}$ against the collection $\mathcal{C}$. Element labels may possibly be followed by a '*'.
- An *ordering list*, each component of which comprises an *order direction* and an element or element attribute (specified by label in $\mathcal{P}$) with values drawn from an ordered domain. The order direction is either ASCENDING or DESCENDING. This ordering list is used to order members of a group for output, based on the values of the component elements and attributes, considered in the order specified.

The output tree S_i corresponding to each group $\mathcal{W}_i$ is formed as follows: the root of S_i has tag `tax_group_root` and two children; its left child ℓ has tag `tax_grouping_basis`, and one child for each element in the grouping basis above, appearing in the same order as in the grouping basis; if a grouping basis item is

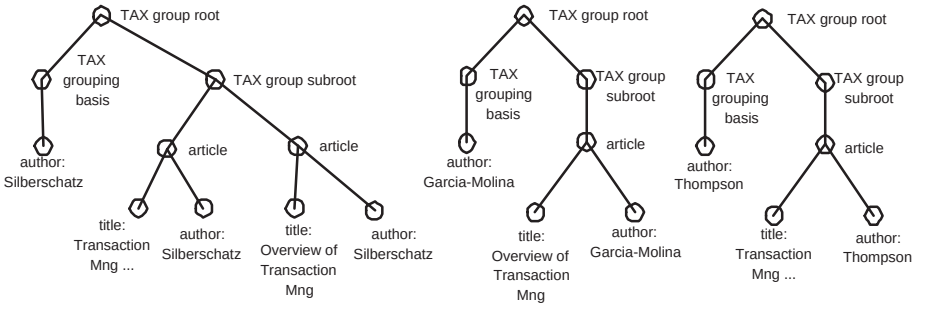

Fig. 3. Grouping the witness trees of Figure 2 by author (`$3.content` in the pattern tree, shown in Figure 1), and ordering each group by descending order of Title (DESCENDING `$2.content`)

`$i` or `$i.attr`, then the corresponding child is a match of this node; if the item is `$i*`, then in addition to the said match, the subtree of the input tree rooted at the matching node is also included in the output; its right child r has tag `tax_group_subroot`; its children are the roots of source trees corresponding to witness trees in $\mathcal{W}_i$, ordered according to the ordering list. Source trees having more than one witness tree will clearly appear more than once.

If the DBLP database has grouping applied to it based on the pattern tree of Figure 1, grouped by author, and ordered (descending) by title, a fragment of the result obtained is shown in Figure 3. Note that articles with two authors appear in two groups, once for each author. In each group, the operation arranges the grouped source trees in decreasing (alphabetical) order of the title subelement.

Grouping, as described above, is already a very powerful operator. We actually have several dimensions in which we can make it even more powerful. For instance, one could use a generic function mapping trees to values rather than an attribute list to perform the needed grouping, one can have a more sophisticated ordering function, and so forth. We do not describe these enhancements in this paper.

4 The Use of Grouping

4.1 Parsing Queries with Grouping

We presented, as Query 1, at the beginning of this paper, an example XQuery expression to compute "For each author in the database, list the author's name and the titles of all articles by that author, grouped inside an authorpubs element". We discuss here how this query is parsed and converted into an algebraic expression in TAX.

Unfortunately a parser cannot detect the logical grouping in the XQuery statement right away. It will "naïvely" try to interpret it as a join. Then a second pass will be necessary to provide a rewrite optimization using TAX's more efficient GROUPBY operator. Below we describe this procedure in detail.

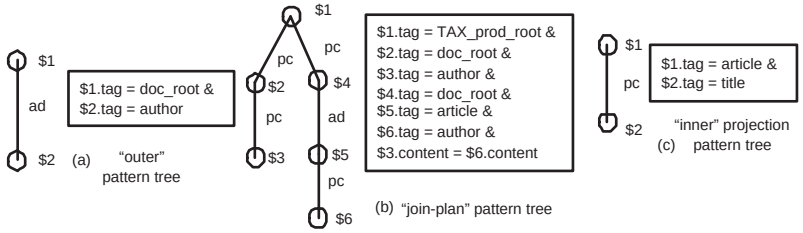

Fig. 4. Pattern trees used during the naïve parse of Query 1

Naïve Parsing

1. The outer combination of FOR/WHERE clauses will generate a pattern tree ("outer" pattern tree). A selection will be applied on the database[4] using this pattern tree; the selection list consists of the bound variables in XQuery. Then a projection[5] is applied using the same pattern tree; the projection list includes all nodes of the tree and has * for each bound variable. Following the projection there will be a duplicate elimination based on the content of the bound variable. For Query 1 the pattern tree is shown in Figure 4.a. The selection list is $2, the projection list is $1 and $2*, and the duplicate elimination is based on $2.content.
2. The RETURN clause will now be processed. Each argument in the return clause is processed at a time. Each argument can create one or multiple pattern trees. Then the appropriate operators will be used taking as input those pattern trees. In the common case a selection and a projection would be used. But aggregate functions may appear here as well etc. At the end, the appropriate stitching of the results will take place. For Query 1 the process is the following.

   {$a}: A pattern tree will be generated containing the author element, corresponding to the already bound variable $a, and the document root. Then a selection and a projection would be applied on the outcome of the "outer" selection using this pattern tree. The selection list is $2 (author) and the projection list $2* (author*).

   Nested FLWR: The procedure for the nested FLWR statement is a little bit more complicated. We will generate one pattern tree for the FOR/WHERE combination and a different one for the RETURN clause.

   (a) The FOR/WHERE clauses will generate a pattern tree that describes a left outer join between all the authors of the database, as selected already and bound to variable $a, and the authors of articles. This pattern tree is shown in Figure 4.b. A left outer join is generated using this pattern tree and applied on the outcome of the

[4] The database is a single tree document

[5] When a projection follows a selection using the same pattern tree, all the ancestor-descendant edges of the tree will be changed to parent-child for the projection.

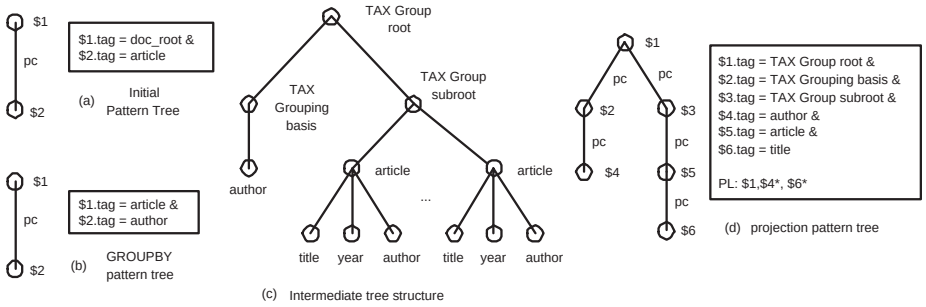

Fig. 5. GROUPBY operator for Query 1. The generated input and the intermediate tree structure

"outer" selection and the database. It uses a selection list \$5. Following this join operation there will be a projection with projection list \$5* and then a duplicate elimination based on articles.

(b) The RETURN clause will be processed one argument at a time. The single argument in this case will generate a pattern tree for the titles as shown in Figure 4.c. Using this pattern tree a selection and then a projection will be applied on the outcome of the previous step. The corresponding selection list is \$2 and projection list \$2*. The output of this step will be returned to the processing of the "outer" RETURN clause.

Stitching: The necessary stitching will take place using a full outer join and then a renaming to generate the tag name for the answer.

Rewriting One can argue that the naive implementation for Query 1 will be inefficient because of the multiple selections over the database and the left outer join used to compute a structural relationship that should already be "known". We next present a rewriting algorithm that transforms the TAX algebra expression described above. The algorithm consists of two phases. Phase 1 detects a grouping query and Phase 2 rewrites the expression using the GROUPBY operator. The rewritten expression can then be used as the basis of an execution plan.

PHASE 1:

1. Check for a left outer join applied on the outcome of a previous selection and the database.
2. Check to see if the left ("outer") part of the join-plan pattern tree is a subset of the right ("inner") part. A tree V_1, E_1 is said to be a subset of a tree V_2, E_2 if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2*$, the transitive closure of E_2.[6]

[6] Each edge e in E_1 may be a parent-child edge, or an ancestor-descendant edge. We may place a mark on each edge of the latter type. Edges in the transitive closure, derived as the composition of two or more base edges, must also have such a mark.

If all the conditions above are TRUE, then we have detected a grouping operator, and we can apply the following rewrite rules.
PHASE 2:

1. Construct an initial pattern tree. The pattern tree is created from the right "inner" subtree of the join plan pattern tree of the naïve parsing. The pattern tree consists of the bound variables and includes their path starting from the document root. The bound variables correspond to the elements in the projection list that of the join plan projection. For Query 1 this pattern tree is seen in Figure 5.a. We apply a selection using this pattern tree with selection list the elements corresponding to the bound variables and a projection with a projection list similar to the selection list. For Query 1 those lists will be $2 and $2* respectively.
2. Construct the input for the GROUPBY operator.
   - The *input pattern tree* will be generated from a subtree of the "inner" pattern tree of naïve parsing. For Query 1 this is shown in Figure 5.b.
   - The *grouping basis* will be generated using the join value of the "join-plan" pattern tree of naïve parsing. For Query 1 this will correspond to the author element or $2.content in the group by pattern tree of Figure 5.b.
   - The *ordering list* will be generated from the projection pattern tree of the inner FLWR statement; only if sorting was requested by the user. So for Query 1, there is an empty ordering list.
3. Apply the GROUPBY operator on the collection of trees generated from step 1. This will create intermediate trees containing each grouping basis element and the corresponding pattern tree matches for it. For Query 1 the tree structure will be as in Figure 5.c.
4. A projection is necessary to extract from the intermediate grouping tree the nodes necessary for the outcome. The projection pattern tree is generated by the projection pattern trees from each argument of the RETURN clauses. For query 1 this is shown in Figure 5.d.
5. After the final projection is applied the outcome consists of trees with an dummy root and the authors associated with the appropriate titles. A rename operator is necessary to change the dummy root to the tag specified in the return clause. This is similar to the rename executed in naïve parsing.

Using an example Let's consider the sample database of Figure 6. Query 1 is executed on this database. Figures 7 and 8 show the generated collections of trees during the naïve parsing phase of the query. TIMBER[23] would typically transform the naïve plan to use the more efficient GROUPBY operator. First, a selection and a projection will be applied on the database using the pattern tree of Figure 5.a as described in phase 2 step 1. This will produce a collection of trees containing all article elements and their entire sub-trees, as in Figure 9. Next the

Please note that for corresponding marked edges in E_1 and E_2, $pc \subseteq ad$, but not $ad \subseteq pc$.

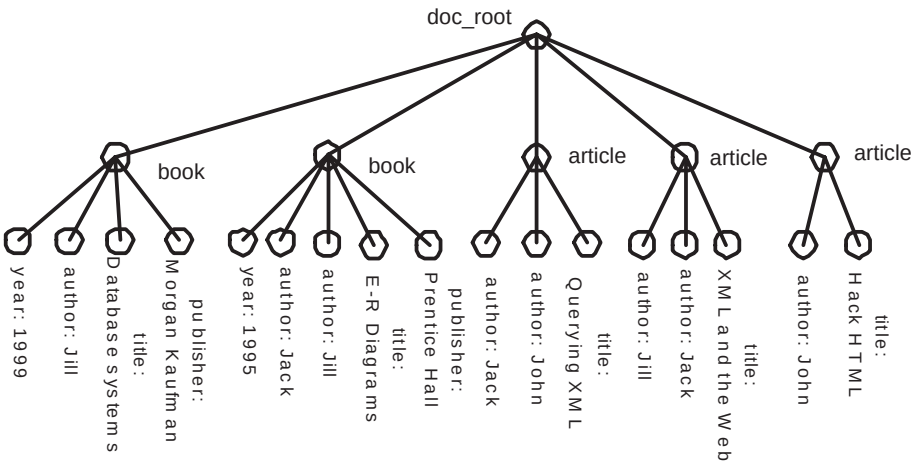

Fig. 6. Sample Database

input pattern tree to be used by the operator will be generated. For Query 1 this is shown at Figure 5.b. The GROUPBY operator (grouping basis : $2.content) will be applied on the generated collection of trees and the intermediate tree structures of Figure 10 are produced. Then the projection is done using the final projection pattern tree from Figure 5.d.

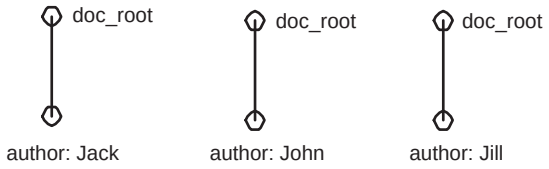

Fig. 7. Applying the selection, projection and duplicate elimination using "outer" pattern tree (Figure 4.a) onto the sample database

4.2 Alternative Query Formulation

Using the set-binding property of the LET clause, Query 1 could equivalently be expressed without nesting, as seen in Query 2. There is no easy way, at the language level, to transform the more common nested expression of Query 1 into the equivalent unnested expression of Query 2. However, at the algebra level, the two expressions are easily seen to be equivalent. Let us walk through an algebraic parsing of Query 2 to note the similarities and differences.

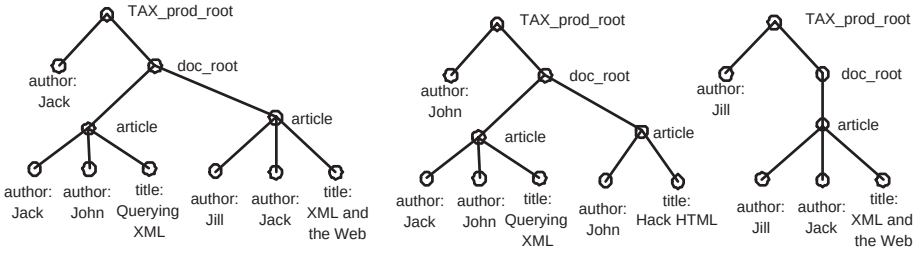

Fig. 8. Generating the left outer join

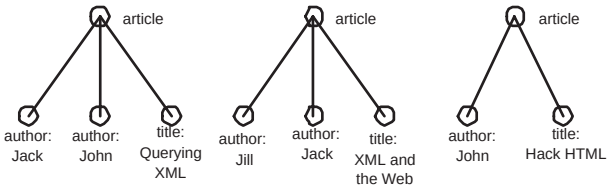

Fig. 9. The collection of trees produced after applying the selection and projection as described in phase 2 step 1 on the sample database of Figure 6

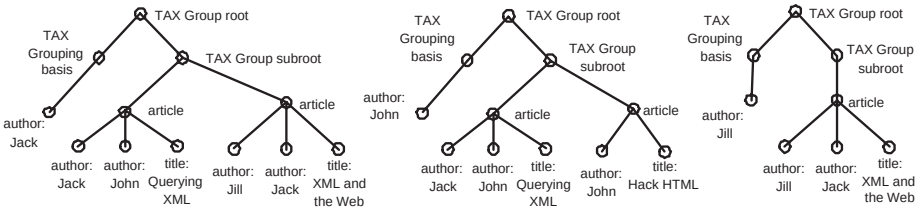

Fig. 10. The intermediate trees produced after applying the GROUPBY operator for Query 1 to the collection of trees of Figure 9

```
FOR $a IN distinct-values(document("bib.xml")//author)
LET $t := document("bib.xml")//article[author = $a]/title
RETURN
   <authorpubs>
      {$a} {$t}
   </authorpubs>
```

Query 2: FLWR with no nesting that groups articles by author.

1. The FOR clause will generate an initial pattern tree, similar with the "outer" pattern tree of the nested query. For Query 2 see Figure 11.a. A selection will be applied on the database using this pattern tree and a selection list that corresponds to the bound variable. For Query 2 the list is $2. Then a projection[7] will be applied on the outcome of the selection, with projection list that corresponds to the bound variable. For Query 2 this is $2*. And a duplicate elimination based on the bound variable element. For Query 2 based on the author element.
2. The LET clause will generate a left outer join pattern tree to be applied on the outcome of the previous step and the database (similar with the left outer join pattern tree of the nested query). For Query 2 this is shown in Figure 11.b.
3. The RETURN clause will be again processed one argument at a time. Appropriate pattern tree(s) will be generated for each argument and a selection and a projection will be applied on the outcome of the previous step. For Query 2 we will have two pattern trees, one for all authors and one for all titles, as seen in Figures 11.c and 11.d.
4. A full outer join is necessary to stitch the return arguments together and then a renaming to include the tag name.

As one can see this algorithm has lots of similarities with the naïve parsing of the nested grouping query. The same kind of pattern trees are generated, with minor differences, such as whether the title node is present in the left outer join pattern tree. After the rewrite optimization, the GROUPBY obtained is identical in both cases.

4.3 Aggregation

The purpose of aggregation is to map collections of values to aggregate or summary values. Common aggregate functions are MIN, MAX, COUNT, SUM, etc. When generating summary values, we should specify exactly where the newly computed value should be inserted, as the content of a new element (or value of a new attribute). More precisely, the aggregation operator **A** takes a collection as input and a pattern $\mathcal{P}$, an aggregate function f_1 and an *update specification*

[7] When a projection follows a selection using the same pattern tree, all the ancestor-descendant edges of the tree will be changed to parent-child for the projection.

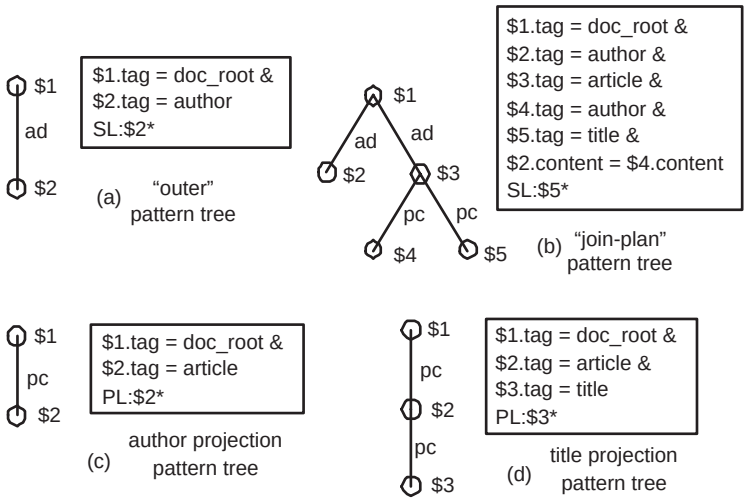

Fig. 11. Grouping expressed without nesting. These are all the selection and projection pattern trees generated during the naive parsing phase

as parameters. The update specification denotes where the aggregate value computed should be inserted in the output trees. The exact set of possible ways of specifying this insertion is an orthogonal issue and should anyway remain an extensible notion. We only give some examples of this specification. E.g., we might want the computed aggregate value to be the last child of a specified node (*after lastChild*(`$i`)), or immediately preceding or following a specified node (*precedes* (`$i`)).

We assume the name of the attribute that is to carry the computed aggregate value is indicated as `aggAttr` $= f_1($`j`.`attr`$)$, or as `aggAttr` $= f_1($`j`$)$, where `aggAttr` is a new name and `$j` is the label of some node in $\mathcal{P}$.

The semantics of the operator $\mathbf{A}_{\texttt{aggAttr}=f_1(\$j.attr),afterlastChild(\texttt{\$i})}(\mathcal{C})$ is as follows. The output contains one tree corresponding to each input tree. It is identical to that input tree except a new right sibling is created, for the node in the output data tree that is the right-most child of the node that matches the pattern node labeled $i in $\mathcal{P}$. This node has content v, where v is the computed aggregate value.

5 Implementation of Grouping

The grouping operations described above, and the TAX algebra of which they are part, have been implemented in the context of the TIMBER[23] native XML database system at the University of Michigan. In this section, we first present a brief overview of TIMBER and then focus on the implementation of grouping specifically.

5.1 System Architecture

TIMBER is built on top of Shore [3], a popular back-end store that is responsible for disk memory management, buffering and concurrency control. XML data, index and metadata are also stored in Shore through Data Manager, Index Manager and Metadata Manager, respectively.

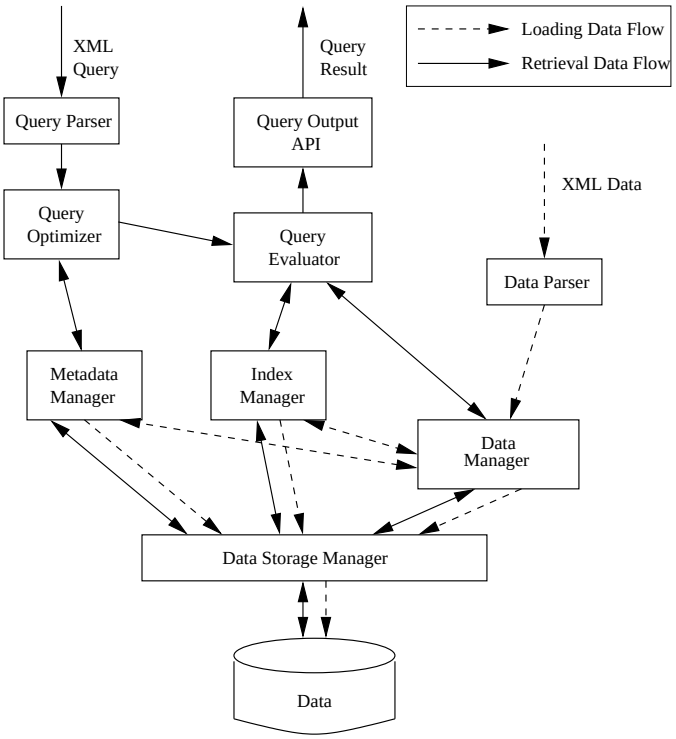

Fig. 12. TIMBER Architecture Overview

The overall architecture of TIMBER is shown in Figure 12. The Query Parser transforms XQuery input into a TAX algebraic expression, performing transformations of the sort described in Sec. 4.2. The Query Optimizer chooses an evaluation plan for the query and hands off to the Query Evaluator. We focus on the evaluation of the grouping operator here. There are two steps: first, the pattern tree has to be matched to develop the requisite node bindings; second, the relevant bound nodes have to be manipulated as per the grouping specification. We consider each step below in turn.

5.2 Pattern Tree Matching

A pattern tree, such as the one is Figure 1 explicitly specifies predicates at nodes that must be satisfied by (candidate) matching nodes and also specifies structural

relationships between nodes that match. Each edge in the pattern tree specifies one such structural relationship, which can either be "parent-child" (immediate containment) or "ancestor-descendant" (containment).

The simplest way to find matches for a pattern tree is to scan the entire database. By and large, a full database scan is not what one would like to perform in response to a simple selection query. One would like to use appropriate indices to examine a suitably small portion of the database. One possibility is to use an index to locate one node in the pattern (most frequently the root of the pattern), and then to scan the relevant part of the database for matches of the remaining nodes. While this technique, for large databases, can require much less effort than a full database scan, it can still be quite expensive.

Experimentally it has been shown [1, 19] that under most circumstances it is preferable to use all the indices available and independently locate candidates for as many nodes in the pattern tree as possible. Structural containment relationships between these candidate nodes is then determined in a subsequent phase, one pattern tree edge at a time. For each such edge, we have a containment "join condition" between nodes in the two candidate sets. We choose pairs of nodes, one from each set, that jointly satisfy the containment predicate. We have developed efficient single-pass containment join algorithms [1] whose asymptotic cost is optimal.

The details of these algorithms is beyond the scope of this paper. The important point to note is that sets of node bindings for pattern trees can be found efficiently. Moreover, these node bindings can be found, in most cases, using indices alone, without access to the actual data. The bindings are represented in terms of node identifiers, obtained from the index look up.

5.3 Identifier Processing

RDBMS implementations of grouping typically rely on sorting (or possibly hashing).[8] We cannot use these implementations directly, since XML grouping does

[8] When an index exists on the (first few) elements of the grouping basis, such an index can be used to perform grouping in relational systems. In most cases, grouping involves access to large parts of the relation, so that access to an unclustered list of indexed entries is not efficient in the absence of additional engineering tricks. In XML, the use of indices is further limited on two accounts. The first is type heterogeneity – an index on value is built over some domain, and there could be many different elements (and even element types) and attributes in the database that are all rolled into one index. The grouping may only be with respect to some of these. For example, to group books by the content of the author sub-element, we require an index on the content of elements representing authors of books, as opposed to authors of articles and authors of reports. The second difficulty with XML value indices is that they only return the identifier of the node with the value in question, whereas we would typically be interested in grouping some other (related) node. For example, an index on book-authors, even if available, is likely to return the author node identifier from which one would have to navigate to the book nodes, which we wish to group.

not necessarily partition the set. One possibility is for us to replicate elements an appropriate number of times, and to tag each replica with the correct grouping variables to use. For example, a two-author book could be replicated to produce two versions of the book node, with one author tagged in each replica as the one to use for grouping purposes. Thereafter standard sorting (or hashing) based techniques may be used.

The difficulty with this approach is that large amounts of data may be replicated early in the process. Particularly, if the required end result is small, for instance because the grouping is followed by aggregation, one would hope that this replication could be avoided.

Our implementation uses a slight variation of the above approach that minimizes these disadvantages. Recall that the grouping basis (the list of variables on the basis of which to group) consists of nodes identified by means of a pattern tree match. The normal pattern tree match procedure will produce all possible tuples of bindings for these grouping variables in the form of witness trees. There is one witness tree per tuple of bindings, and all of these can be obtained using node identifiers only, without access to actual data. For example, the pattern tree of Figure 1 applied to a small subset of the DBLP database produces one witness tree for each book/author pair: there are two witness trees corresponding to a book with two authors – one for each author. See Figure 2.

For elements/attributes in the grouping basis, we need to obtain values to be able to perform the grouping. This requires a data look-up. We populate only the grouping (and sorting) list values, and retain the remainder of the witness tree in identifier form. A sorting based on these list values as key produces the requisite grouping, with each group sorted as specified. Notice that the sorting is performed with minimum information – only a witness tree identifier in addition to the actual sort key.

In the final step of grouping, data can be populated in the grouped and sorted witness trees, as required to produce output. Frequently, this data population is not required, or only partially required, by the subsequent query evaluation operators. For instance, in our running example, we wish to return only titles of books grouped by author, so only the title nodes need be populated with values – the other nodes, book, publisher, date, etc. can all be projected out. A more compelling case is made when the grouping were to be followed by aggregation, as is frequently the case. Suppose we are interested in the count of books written by each author. We can perform the count without physically instantiating the book elements.

6 Experiments

We have shown above that complex nested XQuery expressions can frequently be expressed as single block tree algebra expressions with grouping. Here, we assess the performance benefits of such rewriting. We do so by comparing the performance of the TIMBER implementation of an algebraic expression with grouping and the performance of the same system using a nested loops evaluation

plan obtained through a direct implementation of the corresponding XQuery expression as written. We report results for the group by author query that was introduced as a running example at the beginning of this paper.

We used the Journals portion of the DBLP data set for our experiments. The data loaded comprised 4.6 million nodes, requiring almost 100 MB of storage. We constructed an index on tag-name, so that given a tag, we could efficiently list (by node identifier) all nodes with that tag. We ran our experiments on a Pentium III machine running at 550 MHz. The experiments were run with the database buffer pool size set at 32MB, using a page size of 8 KB. (In other words, even though the machine we used had 256 MB of RAM, only 32 MB of this was available to the query evaluation).

We executed the query in two different ways. The first is a "direct" execution of the XQuery as written. We used the index to identify author nodes, as well as to identify article-author pairs. Then we eliminate duplicates in the former (by looking up the actual data values) and perform the requisite join with the latter. For each author, we output the name (content of the node, which we have already looked up), and for each article found to join with this author, we look up the title, which we output. This process required 323.966 seconds to execute.

The second evaluation plan was the standard TIMBER plan, using the grouping operator. After we have grouped articles by author, we still have to look up the title of each article for output. This plan required 178.607 seconds to execute: a little over half of the direct execution.

Note that both plans require access to the data to look up the content of author and title nodes. Moreover, all of this data is also produced in the output. The content of title nodes is often fairly long. The difference between two different evaluation plans can be highlighted if this common cost were not present. Consider, for instance, the following variant of the preceding query:

```
FOR $a IN distinct-values(document("bib.xml")//author)
LET $t := document("bib.xml")//article[author = $a]/title
RETURN
   <authorpubs>
      {$a} {count($t)}
   </authorpubs>
```

We now seek to output only the count of journal articles for each author rather than list all their titles. The size of output is considerably reduced. Also, the data value look up is now confined to author content: the titles of articles are no longer of interest.

We evaluated this modified query using the two plans described above, with corresponding small changes. Now the direct XQuery evaluation required 155.564 seconds, while the grouping implementation went down to just 23.033 seconds. In other words, the grouping-based implementation was more than 6 times as fast!!

7 Related Work

Several mapping techniques have been proposed [7, 10, 16, 17] to express tree-based XML data to flat tables in a relational schema. Due to the possible absence of attributes and sub-elements, and the possible repetition of sub-elements, XML documents can have a very rich structure. It is hard to capture this structure in a rigid relational table without dividing the document into very small standard "units" that can be represented as tuples in a table. Therefore, a simple XML schema often produces a relational schema with many tables. Structural information in the tree-based schema is modeled by joins between tables in the relational schema. XML queries are converted into SQL queries over the relational tables, and even simple XML queries often get translated into expensive sequences of joins in the underlying relational database. A typical translation [16] of the schema of the DBLP bibliography would map the article elements to a table, and store author elements as tuples in another table. To find the authors of a specified article will then require a join between the two tables. More complex queries will require multiple joins. Grouping is well understood in the relational context, but it is not obvious how to translate grouping at the XQuery level into grouping on the underlying relational tables.

There also are several direct implementations of XML data management, where XML data is not translated into relations [14, 13, 20, 22, 24, 6, 11, 21, 12, 15, 9]. We are not aware of any of these having studied grouping facilities. In fact, many of these are tuple-at-a-time (or navigational) implementations, so that the question of set-oriented grouping does not arise.

8 Conclusion

Grouping is an important operation in relational databases, and made all the more important in the context of XML due to the greater complexity of structure possible. However, this flexibility in structure raises challenges in both the specification and the implementation of grouping.

We have described how the TAX tree algebra can be used to specify potentially involved grouping constructs. We have also shown how queries that appear to be nested in XQuery can be rewritten as a simple query with grouping in TAX.

We have described how grouping is implemented in the TIMBER native XML database system currently being implemented at the University of Michigan. We have demonstrated that the implementation of an explicit grouping operator leads to significant performance benefits over an equivalent nested join query.

References

1. Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE Conf.*, 2002.

2. D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, and P. Rodriguez-Gianolli. ToX - The Toronto XML Engine. *Proc. Intl. Workshop on Information Integration on the Web*, Rio, 2001.
3. M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up Persistent Applications. In *Proc SIGMOD Conf.*, pages 383–394, 1994.
4. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A Query Language for XML. W3C Working Draft. Available from http://www.w3.org/TR/xquery
5. DBLP data set. Available at http://www.informatik.uni-trier.de/ley/db/index.html.
6. L. Fegaras and R. Elmasri. Query Engines for Web-Accessible XML Data. In *Proc. VLDB Conf.*, 2001.
7. D. Florescu and D. Kossman. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
8. H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL Conf.*, Rome, Italy, Sep. 2001.
9. Carl-Christian Kanne, Guido Moerkotte: Efficient Storage of XML Data. Poster abstract in *Proc. ICDE Conf.*, page 198, San Diego, CA, March 2000.
10. M. Klettke, H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *Informal Proc. WebDB Workshop*, pages 151–170, 2000.
11. S. A. T. Lahiri and J. Widom. Ozone: Integrating Structured and Semistructured Data. In *Proc. DBPL Conf.*, Kinloch Rannoch, Scotland, Sep. 1999.
12. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record 26(3)*, pages 54–66, 1997.
13. Microsoft XQuery Language Demo. Online at http://131.107.228.20/xquerydemo/
14. Arnaud Sahuguet. Kweelt: More Than Just "Yet Another Framework to Query XML!". *Proc. SIGMOD Conf.*, Santa Barbara, CA, 2001. Software available from http://db.cis.upenn.edu/Kweelt/.
15. Harald Schoning. Tamino - A DBMS designed for XML. In *Proc. ICDE Conf.*, pp. 149–154, 2001.
16. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.DeWitt, and J.Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. VLDB Conf.* pages 302–314, Edinburgh, Scotland, Sep. 1999.
17. T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Proc. DEXA Conf.*, 1999.
18. Yuqing Wu, Jignesh Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proc. EDBT Conf.*, Prague, Czech Republic, Mar. 2002.
19. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management systems. In *Proc. SIGMOD Conf.*, Santa Barbara, CA, 2001.
20. Tamino Developer Community QuiP, a W3C XQuery Prototype. Available at http://www.softwareag.com/developer/quip.
21. eXcelon Corp. eXcelon XML platform. Available at http://www.exceloncorp.com/platform/extinfserver.shtml.
22. X-Hive Corp. X-Hive/DB. Available at: http://www.x-hive.com.
23. University of Michigan, TIMBER native XML database. Available at http://www.eecs.umich.edu/db/timber/
24. dbXML Group. dbXML Core. Available at: http://www.dbxml.org.