TAX: A Tree Algebra for XML

H. V. Jagadish
University of Michigan
jag@eecs.umich.edu

Laks V. S. Lakshmanan
Concordia University/UBC
laks@cs.concordia.ca

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Keith Thompson
University of Michigan
kdthomps@eecs.umich.edu

Abstract

Querying XML has been the subject of much recent investigation. A formal bulk algebra is essential for applying database-style optimization to XML queries. We develop such an algebra, called TAX (Tree Algebra for XML), 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 develop 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. It forms the basis for the TIMBER XML database system currently under development by us.

1 Introduction

XML has emerged as the *lingua franca* for data exchange, and even possibly for heterogeneous data representation. There is considerable interest in querying data represented in XML. Several query languages, such as XQuery [8], Quilt [7], XML-QL [13], and XQL [28], have recently been proposed for this purpose.

This leads us to the question of implementation. If we expect to have large data sets managed using XML, then we must be able to evaluate efficiently queries written in these XML query languages against these data sets. Experience with the successful relational technology tells us that a formal bulk algebra is absolutely essential for applying standard database style query optimization to XML queries.

An XML document is often viewed as a labeled ordered rooted tree. The DOM [33] application interface standard certainly treats XML documents in this way. There often are, in addition, cross-tree "hyperlinks." In our model, we distinguish between these two types of edges. A similar approach has been adopted in [27]. With this model in mind, in this paper we develop a simple algebra, called *Tree Algebra for XML* (TAX), for manipulating XML data modeled as forests of labeled, ordered trees. The primary challenges we address are: (i) how to permit the rich variety of tree manipulations possible within a simple declarative algebra, and (ii) how to handle the considerable heterogeneity possible in a collection of trees of similar objects (e.g., books). If we look at popular XML query languages, most (including XQuery, which is likely to become the standard) follow an approach of binding variables to tree nodes, and then manipulating the use of these variables with free use of looping constructs where needed. A direct implementation of a query as written in these languages will result in a "nested-loops" execution plan. More efficient implementations are frequently possible — our goal is to devise a bulk manipulation algebra that enables this sort of access method selection in an automated fashion.

We begin in Section 2 by discussing the issues in designing a bulk manipulation algebra for XML. This leads up to our data model in Section 3. A key abstraction in TAX for specifying nodes and attributes is that of the *pattern tree*, presented in Section 4. We describe the TAX operators in Section 5. In Section 6, we show that TAX is complete for relational algebra extended with aggregation, and also establish translation theorems identifying substantial classes of queries expressible in popular XML query languages that can be translated to TAX. TAX has been designed with

an efficient implementation very centrally kept in mind. In Section 7, we discuss a few key issues regarding query evaluation and optimization using TAX, and mention a few key implementation choices made in TIMBER, a native XML database system under development by us. We discuss related work in Section 8 and conclude in Section 9.

2 Design Considerations

A central feature of the relational data model is the declarative expression of queries in terms of algebraic expressions over *collections of tuples*. Alternative access methods can then be devised for these bulk operations. This facility is at the heart of efficient implementation in relational databases.

If one is to perform bulk manipulations on collections of trees, relational algebra provides a good starting point — after all most relational operations (such as selection, set operations, product) are fairly obvious operations one would want to perform on XML databases. The key issue here is what should be the individual members of these collections in the case of XML. In other words, what is the correct counterpart in XML for a relational tuple?

Tree Nodes: One natural possibility is to think of each DOM node (or a tagged XML element, along with its attributes) as the XML equivalent of a tuple. Each element has some named attributes, each with a unique value, and this structure looks very similar to that of a relational tuple. However, this approach has some difficulties. For instance, the defined attributes need not be the same for nodes in a tree, or even for nodes with the same tag name. So the "tuples" are not all drawn from the same domain (cartesian product of attribute domains). They are "incommensurate" and cannot all be placed in a single relation and manipulated meaningfully. More importantly, XML manipulation often uses structural constructs, and element inclusion (i.e., the determination of ancestor-descendant relationship between a pair of nodes in the DOM) is a frequently required core operation. If each node is a separate tuple, then determining ancestor-descendant relationships requires computing the transitive closure over parent-child links, each of which is stated as a join operation between two tuples. This is computationally prohibitive. Clever encodings, such as in [35] can ameliorate this difficulty, but we are still left with a very low level of query expression if these encodings are reflected in the language and data model. Indeed, such encodings should be viewed as implementation techniques for efficient determination of ancestor-descendant relationships, that can be used independent of which data model and algebra we choose.

An alternative data model is to treat an entire XML tree (representing a document or a document fragment) as a fundamental unit, similar to a tuple. This solves the problem of maintaining structural relationships, including ancestor-descendant relationships. However, trees are far more complex than tuples: they have richer structure, and the problem of heterogeneity is exacerbated. There are two routes to managing this structural richness.

Tuples of Subtrees: One route, inspired by the semantics of XML-QL, is to transform a collection of trees into a collection of tuples in a first step of query processing; a sensible way is to use tuples of bindings for variables with specified conditions. Much of the manipulation can then be applied in purely relational terms to the resulting collection of tuples. Trees in the answer can be generated in one final step. However, repeated relational construction and deconstruction steps may be required between semantically meaningful operations, adding considerable overhead. Furthermore, such an approach would lead to limited opportunities for optimization.

Pure Trees: The remaining route is to manage collections of trees directly. This route sidesteps many of the problems mentioned above, but exacerbates the issue of heterogeneity. It also presents a major challenge for defining algebraic operators, in view of the relative complexity of trees compared to tuples. Our central contribution in this paper is a decisive response to this challenge.

We introduce the notion of a *pattern tree*, which identifies the subset of nodes of interest in any tree in a collection of trees. The pattern tree is fixed for a given operation, and hence provides the needed standardization over a heterogeneous set. All algebraic operators manipulate nodes and attributes identified by means of a pattern tree. As such, they can apply to any heterogeneous collection of trees! With this innovation, we show most operators

in relational algebra carry over to the tree domain, with appropriate modifications. We only need to introduce a couple of additional operators to deal with manipulation of the tree structure.

Another important property of XML is that each XML document corresponds to an *ordered* tree. When we query a collection of such ordered documents, it turns out that order is critical in some contexts and immaterial in others. Rather than make a choice at design time, TAX permits the graceful melding, even within a single query, of places where order is important and places where it is not.

3 Data Model

The basic unit of information in the relational model is a tuple. The counterpart in our data model is an ordered labeled tree. A *data tree* is a rooted, ordered tree, such that each node carries data (its label) in the form of a set of attribute-value pairs.

For XML data, each node corresponds to an element and the information content in it represents the attributes of the element, while its children represent its subelements. XML requires that attributes associated with an element be single-valued. We do not require any such assumption for our model, in general. For XML, we assume each node has a special, single valued attribute called `tag` whose value indicates the type of the element. A node may have a `content` attribute representing its atomic value, whose type can be any one of several atomic types of interest: `int, real, string`, etc. The notion of node content generalizes the notion of `PCDATA` in XML documents. For pure `PCDATA` nodes, this tagname could be just `PCDATA`, or it could be a more descriptive tagname if one exists. The notions of ID and IDREFS in XML are treated just like any other attributes in our model. See Figure 1(a) for a sample data tree. Node contents are indicated in parentheses.

We assume each node has a virtual attribute called `pedigree` drawn from an ordered domain.[1] Operators of the algebra can access node pedigrees much like other attributes for purposes of manipulation and comparison. Intuitively, the pedigree of a node carries the history of "where it came from" as trees are manipulated by operators. Since algebra operators do not update attribute values, the pedigree of an existing node is not updated either. When a node is copied, all its attributes are copied, including pedigree. When a new node is created, it has a *null* pedigree. Pedigree plays a central role in grouping, sorting, and duplicate elimination. As we shall show later, appropriate use of this attribute can be valuable for duplicate elimination and grouping, and for inducing/maintaining tree order in query answers. It is useful to regard the pedigree as "document-id + offset-in-document." Indeed, this is how we have implemented pedigree in TIMBER, our implementation of TAX. While pedigree is in some respects akin to a lightweight element identifier, it is *not* a true identifier. For instance, if a node is copied, then both the original and the copy have the same pedigree — something not possible with a true identifier.

A relation in a relational database is a collection of tuples with the same structure. The equivalent notion in TAX is a collection of trees, with similar, not necessarily identical, structure. Since subelements are frequently optional, and quite frequently repeated, two trees following the same "schema" in TAX can have considerable difference in their structure.

A relational database is a set of relations. Correspondingly, an XML database should be a set of collections. In both cases, the database is a set of collections. While this is rarely confusing in the relational context, one frequently has the tendency in an XML context to treat the database as a single set, "flattening out" the nested structure. To fight this tendency, we consistently use the term *collection* to refer to a set of tree objects, corresponding to a relation in a relational database. The whole database, then, is a set of collections.

Relational implementations have found it useful to support relations as multi-sets (or bags) rather than sets. In a similar vein, we expect TAX implementations to implement collections as multi-sets, and perform explicit duplicate elimination, where required.

Each relational algebra operator takes one or more relations as input and produces a relation as output. Correspondingly, each TAX operator takes one or more collections (of data trees) as input and produces a collection as output.

[1] Pedigrees are not shown in our example data trees to minimize clutter.

4 Predicates and Patterns

4.1 Allowable Predicates

Predicates are central to much of querying. While the choice of the specific set of allowable predicates is orthogonal to TAX, any given implementation will have to make a choice in this matter. For concreteness, we mention a representative list of allowable predicates below, with a clear understanding that this list is extensible.

For a node (element) `$i`, any attribute `attr` and value `val` from its domain, the atom `$i.attr` θ `val` is allowed, where θ is one of $=, \neq, >$, etc.[2] As a special case, when `attr` is of type string, a wildcard comparison such as `attr` = "`*val*`", where `val` is a string, is allowed. Similarly, for two nodes `$i` and `$j`, and attributes `attr` and `attr`$'$, the atom `$i.attr` θ `$j.attr`$'$ is allowed. Specifically, the attribute could be the pedigree: predicates of the form `$i.pedigree` θ `c`, `$i.pedigree` θ `$j.pedigree`, where `c` is a constant and θ is $=$ or $\neq$, are also allowed. In addition, atoms involving aggregate operators, arithmetic (e.g., `$i.attr + $j.attr`$'$ `= 0`), and string operations (e.g., `$i.attr = $j.attr`$'$"`terrorism`"), are allowed. Finally, we have predicates based on the position of a node in its tree. For instance, `$i.index = first` means that node `$i` is the first child of its parent. More generally, `index($i, $j) = n` means that node `$i` is the n^{th} node among the descendants of node `$j`. Similarly, `$i before $j` means node `$i` occurs before node `$j`. Both these predicates are based on the preorder enumeration of the relevant data tree.

4.2 Pattern Tree

A basic syntactic requirement of any algebra is the ability to specify attributes of interest. In relational algebra, this is accomplished straightforwardly. Doing so for a collection of trees is non-trivial for several reasons. First, merely specifying attributes is ambiguous: attributes of which nodes? Second, specifying nodes by means of id is impossible, since by design, we have kept the model simple with no explicit notion of object id. Third, identifying nodes by means of their position within the tree is cumbersome and can easily become tricky.

If the collections (of trees) we have to deal with are always homogeneous, then we could draw a tree identical to those in the collection being manipulated, label its nodes, and use these labels to unambiguously specify nodes (elements). In a sense, these labels play a role similar to that of column numbers in relational algebra. However, collections of XML data trees are typically heterogeneous. Besides, we frequently we do not even know (or care about) the complete structure of each tree in a collection: we wish only to reference some portion of the tree that we care about. Thus, we need a simple, but powerful means of identifying nodes in a collection of trees.

We solve this problem using the notion of a *pattern tree*, which provides a simple, intuitive specification of nodes and hence attributes of interest. It also is particularly well-suited to graphical representation.

Definition 4.1 (Pattern Tree) Formally, a *pattern tree* (pattern for short) is a pair $\mathcal{P} = (T, F)$, where $T = (V, E)$ is a node-labeled and edge-labeled tree such that:

- each node in V has a distinct integer[3] as its label;
- each edge is either labeled `pc` (for parent-child) or `ad` (for ancestor-descendant).
- F is a formula, i.e. a boolean combination of predicates applicable to nodes. ∎

While the formal semantics of patterns are given in the next subsection, here we give some examples. Figure 1(b) shows a pattern that asks for books published before 1988 and having at least one author. The edge label `pc` indicates that `year` must be a direct subelement of `book`, while the edge label `ad` indicates that `author` could be any nested descendant subelement. As another example, the pattern in Figure 1(c) asks for books published by a publisher

[2] We also allow the variants `$i._` = `val` and `$i.attr` = `_` ; the former says `val` appears as a value of some attribute, while the latter says the attribute `attr` is defined for node `$i`.

[3] Labels are denoted `$i`, for integer i.

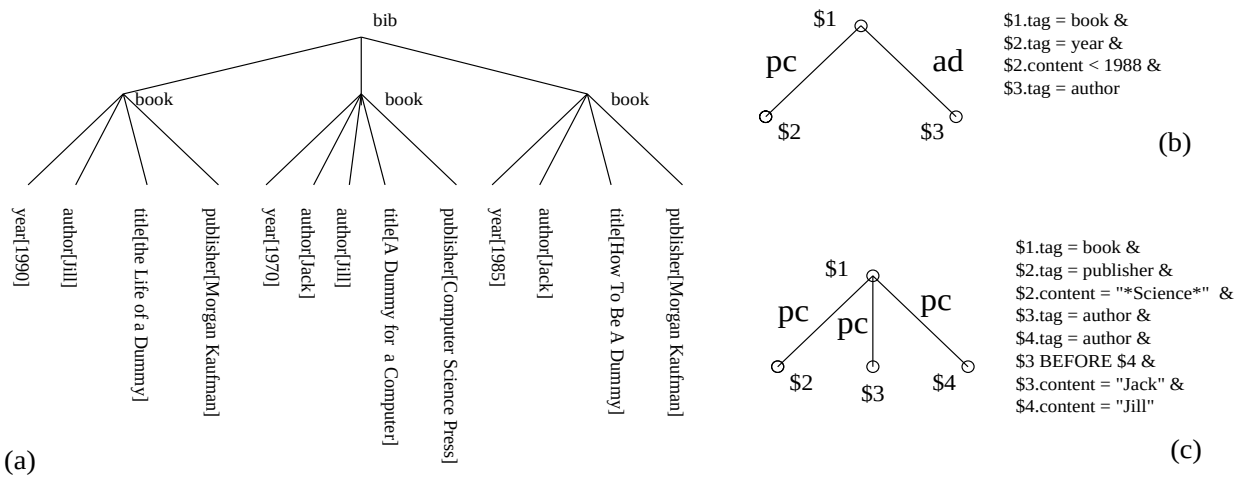

Figure 1: (a) A one-tree XML database, and (b),(c) Two pattern trees

whose name contains the string "Science" and authored by Jack and Jill in that order. In both examples, see how the tree and the formula F interact.

We have chosen to allow ancestor-descendant (`ad`) edges, in addition to the basic parent-child (`pc`) edges, in a pattern tree because we believe that one may often wish to specify just such a relationship without involving any intervening nodes, a feature commonly found in XML query languages. We note that additional constructs are possible. For instance, we could introduce edges that are labeled by constraints, to obtain the full functionality of regular expressions. Any such extension to the definition of a pattern tree is orthogonal to the definition of the TAX operators below.

Pattern trees in TAX also permit attributes of nodes to be compared with other attributes of (other) nodes, analogously to selection predicates in relational algebra permitting different attributes to be compared. See, for instance Figure 1(c), where the position of two nodes is compared using the `BEFORE` predicate.

4.3 Witness Tree

A pattern tree $\mathcal{P} = (T, F)$ constrains each node in two ways. First, the formula F may impose value-based predicates on any node. Second, the pattern requires each node to have structural relatives (parent, descendants, etc.) satisfying other value-based predicates specified in F. Of these, the value-based predicates are in turn based on the allowable set of atomic predicates applicable to pattern tree nodes.

Formally, let $\mathcal{C}$ be a collection of data trees, and $\mathcal{P} = (T, F)$ a pattern tree. An *embedding* of a pattern $\mathcal{P}$ into a collection $\mathcal{C}$ is a total mapping $h : \mathcal{P} \rightarrow \mathcal{C}$ from the nodes of T to those of $\mathcal{C}$ such that:

- h preserves the structure of T, i.e. whenever (u, v) is a `pc` (resp., `ad`) edge in T, $h(v)$ is a child (resp., descendant) of $h(u)$ in $\mathcal{C}$.
- The image under the mapping h satisfies the formula F.

Let $h : \mathcal{P} \rightarrow \mathcal{C}$ be an embedding and let u be a node in T and v a node in $\mathcal{C}$ such that $v = h(u)$. Then we say the data tree node v *matches* the pattern node u (under the embedding h). Note that an embedding need *not* be 1-1, so the same data tree node could match more than one pattern node.

Note also that we have ignored order among siblings in the pattern tree as we seek to embed it in a data tree. Siblings in a pattern tree may in general be permuted to obtain the needed embedding. We have chosen to permit this because such queries seemed to us to be more frequent than queries in which the order of nodes in the pattern tree is material. Moreover, if maintaining order among siblings is desired, this is easily accomplished through the

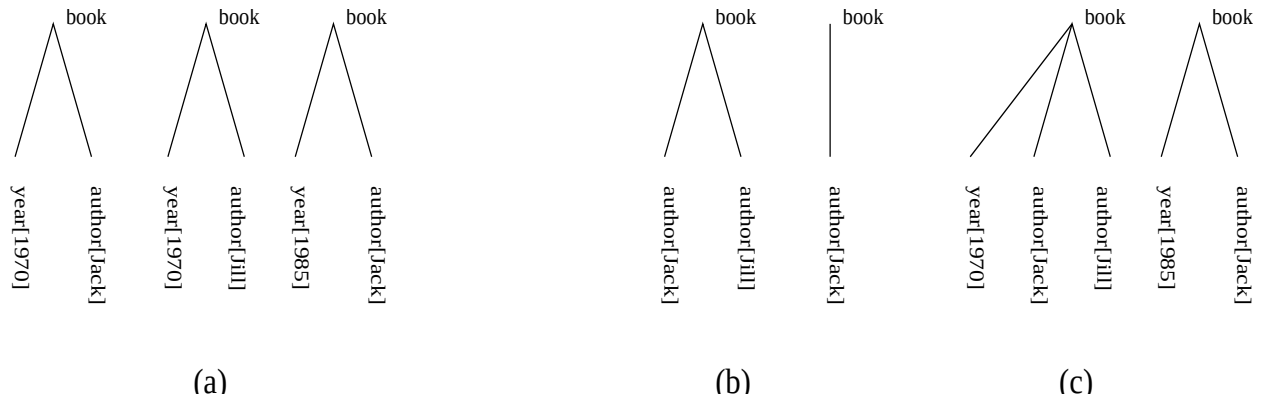

Figure 2: Results of various operations applied to the database of Figure 1(a)

use of ordering predicates (such as `BEFORE`), and can even be applied selectively. For example, Figure 1(c) specifies a pattern that seeks books with authors `Jack` and `Jill`, with `Jack` appearing before `Jill`, and having a publisher ``*Science*'', though we do not care whether the publisher subelement of book appears before or after the author subelements.

We next formalize the semantics of pattern trees using a notion of witness trees. Each embedding of a pattern tree into a database induces a *witness tree* of the embedding:

Definition 4.2 (Witness Tree) Let $\mathcal{C}$ be a collection of data trees, $\mathcal{P} = (T, F)$ a pattern tree, and $h : \mathcal{P} \rightarrow \mathcal{C}$ an embedding. Then the *witness tree* associated with this is the data tree, denoted $h^{\mathcal{C}}(\mathcal{P})$ defined as follows:

- a node n of $\mathcal{C}$ is present in the witness tree if $n = h(u)$ for some node u in the pattern $\mathcal{P}$, i,e. n matches some pattern node under the mapping h.
- for any pair of nodes n, m in the witness tree, whenever m is the closest ancestor of n in $\mathcal{C}$ among those present in the witness tree, the witness tree contains the edge (m, n).
- the witness tree preserves the order on the nodes it retains from $\mathcal{C}$, i.e. for any two nodes in $h^{\mathcal{C}}(\mathcal{P})$, whenever m precedes n in the preorder node enumeration of $\mathcal{C}$, m precedes n in the preorder node enumeration of $h^{\mathcal{C}}(\mathcal{P})$ as well.

Let $I \in \mathcal{C}$ be the data tree such that all nodes of the pattern tree T map to I under h. We then call I the *source tree* of the witness tree $h^{\mathcal{C}}(\mathcal{P})$. We also refer to $h^{\mathcal{C}}(\mathcal{P})$ as the *witness tree* of I under h. ∎

The meaning of a witness tree should be straightforward. The nodes in an instance that satisfy the pattern are retained and the original tree structure is restricted to the retained nodes to yield a witness tree. If a given pattern tree can be embedded in an input tree instance in multiple ways, then multiple witness trees are obtained, one for each embedding. For example, Figure 2(a) shows witness trees resulting from embedding the pattern of Figure 1(b) into the database of Figure 1(a) in three different ways. Thus, the same book appears in two different witness trees, once for each possible author node binding.

4.4 Tree Value Function

Given a collection of trees, we would like to perform ordering and grouping operations along the lines of `ORDERBY` and `GROUPBY` in SQL. In fact, ordering is required if (ordered) trees are to be constructed from (unordered) collections.

However, we once again have to take into account the possible heterogeneity of structure in a collection of trees, making it hard to specify the nodes at which to find the attributes of interest. We solve this problem in a rather

general way, by proposing the notion of a *tree value function* (TVF) that maps a data tree (typically, source trees of witness trees) to an ordered domain (such as real numbers). The exact nature of this TVF is orthogonal to the algebra, although we assume it is efficiently computable. A simple example tree value function might map a tree to the value of an attribute at a node (or a function of the tuple of attribute values associated with one or more nodes) in the tree (identified by means of a pattern tree); an example using TVFs is presented in Section 5.5. Thus, while pattern trees are used in all operators of our algebra, tree value functions are useful in performing grouping and ordering.

5 The Operators

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 notions of pattern tree and tree value function introduced in the preceding section play a pivotal role in many of the operators.

5.1 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.

To appreciate this point, consider selecting books that were published before 1988 from a collection of books. Let it generate a subset of the input collection, as in relational algebra. But if the input collection comprises a single bibliography data tree with book subtrees, as in Figure 1(a), the selection output would return the original data tree, leaving no clue about *which* book was published before 1988.

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. Formally, the output $\sigma_{\mathcal{P},\mathrm{SL}}(\mathcal{C})$ of the selection operator is a collection of trees, one per embedding of $\mathcal{P}$ into $\mathcal{C}$. The output tree associated with an embedding $h : \mathcal{P} \rightarrow \mathcal{C}$ is defined as follows.

- A node u in the input collection $\mathcal{C}$ belongs to the output iff u matches some pattern node in $\mathcal{P}$ under h, or u is a descendant of a node v in $\mathcal{C}$ which matches some pattern node w under h and w's label appears in the adornment list SL.
- Whenever nodes u, v belong to the output such that among the nodes retained in the output, u is the closest ancestor of v in the input, the output contains the edge (u, v).
- The relative order among nodes in the input is preserved in the output, i.e. for any two nodes u, v in the output, whenever u precedes v in the preorder enumeration of $\mathcal{C}$, u precedes v in the preorder enumeration of the output.

Contents of all nodes, including pedigrees, are preserved from the input. As an example, let $\mathcal{C}$ be a collection of book elements in Figure 1(a), and let $\mathcal{P}$ be the pattern tree in Fig 1(b). Then $\sigma_{\mathcal{P},\mathrm{SL}}(\mathcal{C})$ produces exactly the collection of trees in Fig 2(a) if the adornment list SL is empty. On the other hand, if SL includes $1, then the entire subtree is retained for each book (node $1) in the result.

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. See Section 6.1. If we desire only to return

entire trees, from the input collection, that match the selection predicate, then all we have to do is to add a new root node labeled `$1` to the pattern tree, with an `ad` edge to the previous root, and take a conjunction of the previous formula F with `isRoot($1)`[4], and to place `$1` in the adornment list SL.

5.2 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 $*$. The output $\pi_{\mathcal{P},PL}(\mathcal{C})$ of the projection operator is defined as follows.

- A node u in the input collection $\mathcal{C}$ belongs to the output iff there is an embedding $h : \mathcal{P} \rightarrow \mathcal{C}$ such that u matches some pattern node in $\mathcal{P}$ whose label appears in the projection list PL, or u is a descendant of a node v in $\mathcal{C}$ which matches some pattern node w and w and w's label appears in the projection list PL with a "*".
- Whenever nodes u, v belong to the output such that among the nodes retained in the output, u is the closest ancestor of v in the input, the output contains the edge (u, v).
- The relative order among nodes is preserved in the output, i.e., for any two nodes u, v in an output tree, whenever u precedes v in the preorder enumeration of $\mathcal{C}$, u precedes v in the preorder enumeration of the output tree.

Contents of all nodes, including pedigrees, are preserved from the input. As an example, suppose we use the pattern tree of Figure 1(b) and projection list {`$1,$3`}, and apply a projection to the database of Figure 1(a). Then we obtain the result shown in Figure 2(b).

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. For instance, the pattern tree of Figure 1(b) and projection list {`$1,$2,$3`}, used as parameters to a projection of the database of Figure 1(a), results in Figure 2(c). Compare this with the selection result shown in Figure 2(a) for the same pattern tree and database.

5.3 Product

The product operation takes a pair of collections $\mathcal{C}$ and $\mathcal{D}$ as input and produces an output collection corresponding to the "juxtaposition" of every pair of trees from $\mathcal{C}$ and $\mathcal{D}$. More precisely, $\mathcal{C} \times \mathcal{D}$ produces an output collection as follows.

- for each pair of trees $T_i \in \mathcal{C}$ and $T_j \in \mathcal{D}$, $\mathcal{C} \times \mathcal{D}$ contains a tree, whose root is a new node, with a tag name of `tax_product_root`, a null pedigree, and no other attributes or content; its left child is the root of T_i, while its right child is the root of T_j.

[4] Technically, it is unnecessary to assume `isRoot` as a primitive predicate. It can be expressed by subtracting nodes with parents from the set of all nodes. However, expressing it directly improves both readability and efficiency.

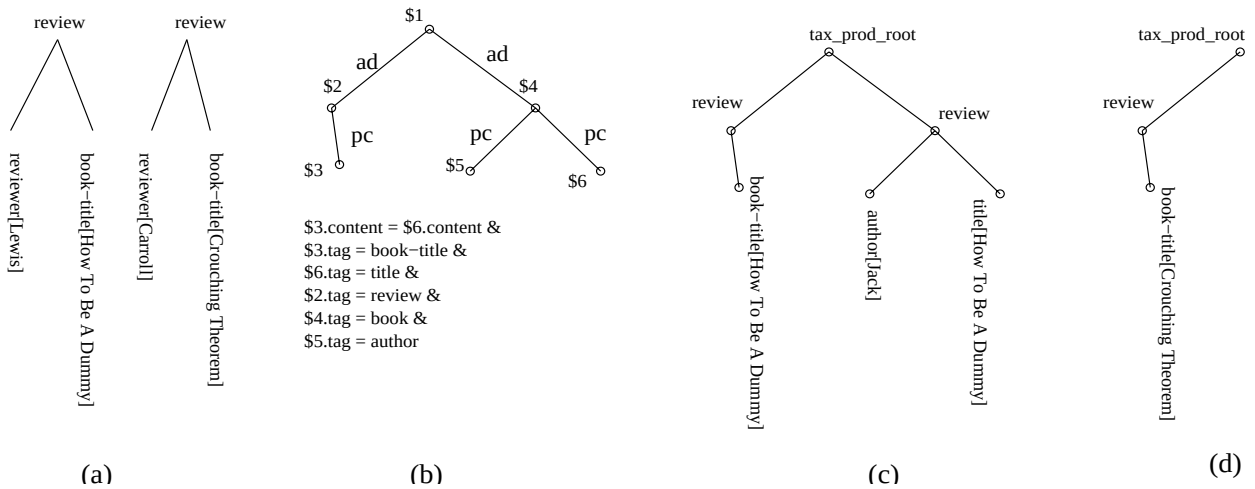

Figure 3: (a) An input collection, (b) A pattern tree, (c) A Join Result, and (d) An additional left outerjoin result

- for each node in the left and right subtrees of the new root node, all attribute values, including pedigree, are the same as in the input collections.

The choice of a null pedigree for the newly created root nodes reflects the fact that these nodes do *not* have their origins in the input collections. Since data trees are ordered, $\mathcal{C} \times \mathcal{D}$ and $\mathcal{D} \times \mathcal{C}$ are not the same. This departure from the relational world is justified since order is irrelevant for tuples but important for data trees which correspond to XML documents.

As in relational algebra, join can be expressed as product followed by selection. For example, the reviews collection of Figure 3(a), joined with the books collection of Figure 1(a), by taking the product and applying a selection with the pattern shown in Figure 3(b), yields the result shown in Figure 3(c).

We can also derive other operators. For instance, the left outerjoin of the same collections, with the same conditions, results in a collection that includes the tree of Figure 3(d) in addition to Figure 3(c). We define these formally below.

Definition 5.1 (Join and Outerjoin) Let $\mathcal{C}_1, \mathcal{C}_2$ be any collections of trees and $\mathcal{P}_i = (T_i, C_i), i = 1, 2$ be any two patterns. Let $\circ^i, i = 1, 2$ be either selection or projection and $L_i, i = 1, 2$ interpreted either as selection list SL or projection list PL depending on what $\circ^i$ is. Then the *join operation* $\circ^1_{\mathcal{P}_{1,L_1}}(\mathcal{C}_1) \bowtie_{\text{newTag};\$i.attr1=\$j.attr2} \circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)$, where \$i refers to a node label in T_1 and \$j to one in T_2, has the following semantics: for each pair of witness trees obtained from the operations $\circ^i_{\mathcal{P}_{i,L_i}}(\mathcal{C}_i), i = 1, 2$, say W_1, W_2, whenever the match of the node \$i, say w_1 in W_1 and the match of node \$j, say w_2 in W_2 are such that $w_1.attr1 = w_2.attr2$ holds then the output collection contains a tree whose root is of tag newTag; its left child is the root of W_1 and right child is the root of W_2.

The *left outerjoin* of the above expressions is defined as follows: $\circ^1_{\mathcal{P}_{1,L_1}}(\mathcal{C}_1) \mathbin{⟕}_{\text{newTag};\$i.attr1=\$j.attr2} \circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)$, where \$i refers to a node label in T_1 and \$j to one in T_2, has the following semantics: for every witness tree W_1 obtained from the operation $\circ^1_{\mathcal{P}_{1,L_1}}(\mathcal{C}_1)$, the output contains one or more trees, determined as follows:

- if for no witness tree W_2 obtained from the operation $\circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)$ the matches w_1 of node \$i in W_1 and w_2 of node \$j in W_2 do not satisfy the condition $w_1.attr1 = w_2.attr2$, then the output contains just one tree corresponding to W_1; its root is of tag newTag and its only child is the root of W_1.
- if the above does not hold, then for every witness tree W_2 obtained from $\circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)$, whenever the matches w_1 of node \$i in W_1 and w_2 of node \$j in W_2 satisfy the condition $w_1.attr1 = w_2.attr2$, then the output contains a tree; its root is of tag newTag, its left chidl is the root of W_1 and the right child is the root of W_2.

The following lemma shows join and left outerjoin are indeed derived operators.

Lemma 5.1 (Derived Operators) : Both join and left outerjoin can be simulated in TAX.

Proof: First, consider join. Construct a new tree pattern as follows. It has a root and has an isomorphic copy of T_1 as its left child and an isomorphic copy of T_2 as its right child. Assign distinct node labels. The associated formula is the cojunction of C_1 and C_2 together with the condition \$i.attr1 = \$j.attr2, where \$i, \$j are the node labels of the ndoes involved in the join condition. Call the resulting pattern $\mathcal{P}$. Then the join is equivalent to the expression $\sigma_{\mathcal{P};SL:\{\}}[\circ^1_{\mathcal{P}_{1,L_1}}(\mathcal{C}_1) \times_{\text{newTag}} \circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)]$. Left outerjoin can now be simulated by projecting $[\circ^1_{\mathcal{P}_{1,L_1}}(\mathcal{C}_1) \times_{\text{newTag}} \circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)] - [\circ^1_{\mathcal{P}_{1,L_1}}(\mathcal{C}_1) \bowtie_{\text{newTag};\$i.attr1=\$j.attr2} \circ^2_{\mathcal{P}_{2,L_2}}(\mathcal{C}_2)]$ on nodes corresponding to the copy of T_1 and unioning it with the join. ∎

5.4 Set Operations

As in the relational model, we fall back on set theory for set union, intersection and difference. The only issue is to specify when two elements (data trees) should be considered identical. Since our trees are ordered, obtaining a correspondence between nodes is straightforward. We then require that all attributes at corresponding nodes, including tag, pedigree, and content, be identical. Formally, two data trees T_1, T_2 are *equal* iff there exists an isomorphism $\iota : T_1 \rightarrow T_2$ between the two sets of nodes that preserves edges and order, and furthermore, for every value-based atom of the form "`attribute` θ `value`", the atom is true at a node u in T_1 iff it is true at node $\iota(u)$ in T_2. Given this notion of (deep) equality, union, intersection, and difference are defined in the standard way. Multi-set versions of these operations are also possible. In particular, for union, we could define the result multiplicity based on *sum* or *max*.

5.5 Grouping

Unlike in the relational model, we separate grouping and aggregation. The rationale is that grouping has a natural direct role to play for restructuring data trees, orthogonally to aggregation. It is worth noting that such an approach has also been taken [2] in some recent proposals for an OLAP algebra.

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 book elements grouped by title. We may wish to group this collection by author, thus generating subsets of book 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.

In relational grouping, it is easy to specify the grouping attributes. In our case, we will need to use a tree value function for this purpose. 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 function* that partitions the set $\mathcal{W}$ of witness trees of $\mathcal{P}$ against the collection $\mathcal{C}$. Typically, this grouping function will be instantiated by means of a *grouping list* that lists elements (by label in $\mathcal{P}$), and/or attributes of elements, whose values are used to obtain the required partition. The default comparison of element values is "shallow", ignoring sub-element structure. Element labels in a grouping list may possibly be followed by a '*', in which case not just the element but the entire sub-tree rooted at this element is matched.
- An *ordering* tree value function *orfun* that maps data trees to an ordered domain. This function is used to order members of a group for output, in the manner described below.

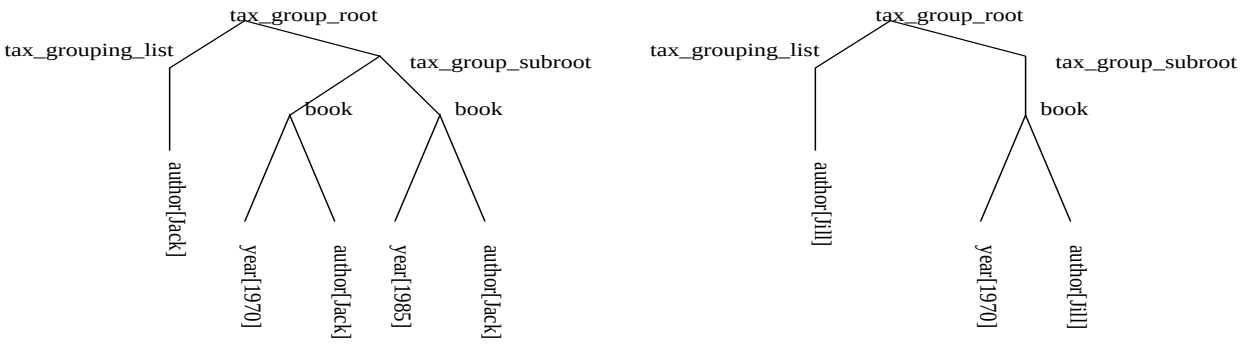

Figure 4: Grouping the witness trees of Fig 2(a) by Author (`$3.content` in the pattern tree, shown in Figure 1b), and ordering each group by year (*orfun* $\equiv$ `$2.content`)

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`, a null pedigree and two children; its left child ℓ has tag `tax_grouping_basis`, a null pedigree, and a sub-tree rooted at this node that captures the grouping basis; its right child r has tag `tax_group_subroot`, a null pedigree; its children are the roots of source trees corresponding to witness trees in $\mathcal{W}_i$, arranged in increasing order w.r.t. the value $orfun(T_j)$, I_j being the source tree associated with the witness tree T_j. Source trees having more than one witness tree will appear more than once in the output – once in each group corresponding to each witness tree.

When a grouping operation is performed, the result should include not just a bunch of groups, but also "labels" associated with each group identifying the basis for creation of this group. (In relational systems, this is the set of grouping attributes for the group, which will be retained separately even as the remaining attributes of tuples in the group are condensed through aggregation). A generic grouping basis function must specify the manner in which this information is to be retained, under the `tax_grouping_basis` node of the result. In the typical case of a grouping list being used to partition, the grouping list can also be applied as a projection list parameter to obtain a projection of the source trees associated with each groups. These projections, by definition, must all be identical within a group, except for their pedigree. By convention, we associate the least of the pedigree values for each node, and eliminate the rest. The result is made a child of the `tax_grouping_basis` node. If the projection returns a forest, order is preserved among the trees in this forest.

Consider the database of Figure 2(a). Apply grouping to it based on the pattern tree of Figure 1(b), grouped by author, and ordered by year. The result is shown in Figure 4. If this grouping had been applied to an XML database consisting of one tree for each book in the example database of Figure 1(a), one of the books (published in 1970) would appear in two groups, one for each of the authors.

A few words regarding the way collections of source trees are partitioned are in order. For every node label of the form `$i` in the grouping list, we use a shallow notion of equality: two matches of this node are equal provided their contents (set of attribute value pairs, except for pedigree) are identical. For every node label of the form `$i*` in the grouping list, we use a deep notion of equality. Under this, two matches of this node are equal provided there is an isomorphism between the subtrees rooted at these matching nodes, that preserves order and node contents (except for pedigree). Deep equality is also used as a basis for defining set operations (see Section 5.4), but there pedigree attributes are also considered. In short, equality can be deep or shallow, and it can be by value (without pedigree) or by node identity (using pedigree). The appropriate notion should be used in each circumstance.

Duplicate Elimination by Value Due to the presence of the pedigree attribute, two distinct nodes in the input, even if identical in value, are not considered duplicates for purposes of set operations.

However, there is often the need to eliminate duplicates by value of (specified) attributes. For example the `distinct` operator in XQuery would require it. We can show

Lemma 5.2 (Duplicate Elimination) : Duplicate elimination of nodes by value can be expressed in TAX. ∎

Proof: The basic idea is to group the given collection of data trees based on the root element, and then delete the right subtree of the root node in each tree in the resulting collection. More precisely, perform grouping using the pattern ($1*, *isRoot*($1)$), and then do a projection with the pattern

```
($1   ,   $1.tag=tax_group_root &   )
  |       $2.tag=tax_grouping_basis
 $2
  |
 $3
```

and the projection list PL = ($3*). It is easy to see that the resulting collection will contain exactly one tree from each set of identical (w.r.t. deep equality) trees in the input collection. ∎

Sorting ORDERBY is a useful operator in SQL, used to present results in a sorted order. Since sets are unordered by definition, this operator can only be applied to the output in relational algebra. In TAX, collections are unordered. However, trees themselves are ordered. Frequently, the result of a query is composed into a (XML) tree, thereby imposing an order on it.

The grouping operator of TAX is used for both ordering a collection and converting it into a tree. To accomplish this, use an empty grouping basis, so that all trees in the input collection map to a single output group. The result is a single tree in the output, with each input tree a child of the tax_group_subroot. The ordering function specifies their relative order. The desired final result is now obtained by projecting away the tax_group_root and the tax_grouping_basis.

If a query specifies an explicit sort order, based on the values of selected elements/attributes, this can be stated in the ordering function, identifying the appropriate elements/attributes with the pattern tree in the usual manner. If a query requires that the output be ordered not on the basis of value, but on the basis of position in the input document, then one can use node pedigrees for this purpose. The ordering function can use a term representation of each tree, using pedigrees for each node, and sort these lexicographically.

5.6 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. However, the exact choice is orthogonal to the algebra. When generating summary values, we should specify exactly where the newly computed value should be inserted, under what tag and/or as value of which 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* 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*($i)), or immediately preceding or following a specified node (e.g. *precedes* ($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 aggregation operator $\mathbf{A}_{\texttt{aggAttr}=f_1(\$j.attr),afterlastChild(\$i)}(\mathcal{C})$ 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 the attribute-value pairs {(tag, tax_aggNode), (aggAttr, v)}, where v is the computed aggregate value. Not being a node in the input, this node has null pedigree.

Often, one is interested not just in the (extremum) aggregate value of some attribute but in the object that possesses that attribute. In relational algebra, the corresponding tuple(s) can be returned by using a join between the computed aggregate value and the original relation. The same strategy can also be used in TAX. However, one can do better since the grouping operator preceding the aggregation would already have performed the necessary "join". An appropriate selection, after the aggregation, will do the trick.

5.7 Renaming

Renaming produces a collection isomorphic to an input collection, while changing the name of specified attributes, or the value of the `tag` attribute, of specified nodes in each input tree. This is analogous to relational renaming where attributes of tuples in a relation are renamed. We need the following notion in the definition of renaming. Let $\mathcal{P}$ be a pattern. Then a renaming specification (RS) is a sequence of expressions of the form `$i:oldName` $\leftarrow$ `newName`, where `$i` is one of the node labels appearing in $\mathcal{P}$, and `oldName`, `newName` are any identifiers. Renaming takes a collection $\mathcal{C}$ as input, and a pattern tree $\mathcal{P}$ and a renaming specification RS as parameters, and generates an output collection $\rho_{\mathcal{P},\text{RS}}(\mathcal{C})$ as output, as follows.

- Every node in $\mathcal{C}$ that matches some pattern node labeled `$i` in $\mathcal{P}$, under some embedding, is marked `i`.
- For every tree in the collection $\mathcal{C}$, $\rho_{\mathcal{P},\text{RS}}(\mathcal{C})$ contains an isomorphic tree, with every attribute, including the pedigree, of every node being identical to that of the corresponding node in $\mathcal{C}$.
- Whenever a node u in the output corresponds to an input node marked `i` and the pattern node labeled `$i` in $\mathcal{P}$, and RS contains the expression `$i:oldName` $\leftarrow$ `newName`, then:
  - if the tag of u is `oldName`, then it is changed to `newName`.
  - if u contains an attribute `oldName`, then it is changed to `newName`.

Renaming is essentially analogous to that in relational algebra. Note the static semantics of renaming: first, every node that is potentially affected by renaming (via any embedding) is marked; then the `newName` values are assigned to the RHS of the relevant expressions, and only then the appropriate changes are made in every marked node. This is needed to ensure that renaming does not have a "cascade" effect (e.g., with RS containing `$i:name1` $\leftarrow$ `name2` and `$j:name2` $\leftarrow$ `name3`, and a node u matches `$i` and `$j` via different embeddings), in the sense that tag names or attributes are not changed more than once and the output is unambiguous. "Reserved" attribute names, such as `tag` and `pedigree`, may not be renamed. Due to multiple embeddings of the pattern tree, it is possible for an input node to get marked more than once. If the renaming operations due to these multiple markings clash (e.g., when RS contains `$i:name1` $\leftarrow$ `name2` and `$j:name1` $\leftarrow$ `name3`), then the renaming specification is ambiguous, and a TAX implementation may choose to resolve the ambiguity as it sees fit. One option is to treat renaming as a noop in this case.

5.8 Reordering

Often we might wish to change the order of children of a node or might want to regroup subelements. For instance, in a bibliography we may want the book elements grouped by publisher, and secondarily by title.

The *reordering* operator ϱ takes a collection $\mathcal{C}$ as an operand, and a pattern $\mathcal{P}$, a tree value function f, and a reorder list RL as parameters. The reorder list is a list of node labels appearing in $\mathcal{P}$. The semantics of reordering is as follows. Suppose for simplicity there is a unique node u in $\mathcal{C}$ that matches any pattern node whose label appears in RL. Let $T_1, \ldots, T_k$ be the child subtrees of u in $\mathcal{C}$. Then they are reordered in increasing order of $f(T_i)$. If f maps different child subtrees to the same number, TAX implementation is free to choose any convenient way of ordering the children.

Suppose now there is more than one input node that matches a pattern node whose label appears in RL. Let R be the set of all such nodes. Sort R so that a descendant always precedes an ancestor. Then we apply the reordering above to the nodes R in the order in which they appear after the sorting. Pedigrees are unaffected by reordering.

As an example, suppose $\mathcal{C}$ is a collection of books grouped by author, where each tree contains all books written by an author. We might choose to order the book subtrees by their year of publication or their price or the number of book pages. All of these are readily handled by invoking the reorder operator ϱ with a reorder list that identifies the book nodes in the pattern and a tree value function that maps a book tree to its year, price, or number of pages, as appropriate. The following lemma can be proved by routine induction.

Lemma 5.3 (Idempotence of Reordering) : Let $\mathcal{P}$ be a pattern, f a tree value function, such that f applied to a subtree rooted at a node is a function of the attribute-value pairs at that node, and RL a reorder list. Then for any collection, $\varrho_{\mathcal{P},f,\text{RL}}(\varrho_{\mathcal{P},f,\text{RL}}(\mathcal{C})) = \varrho_{\mathcal{P},f,\text{RL}}(\mathcal{C})$, i.e. reordering is an idempotent operator. ∎

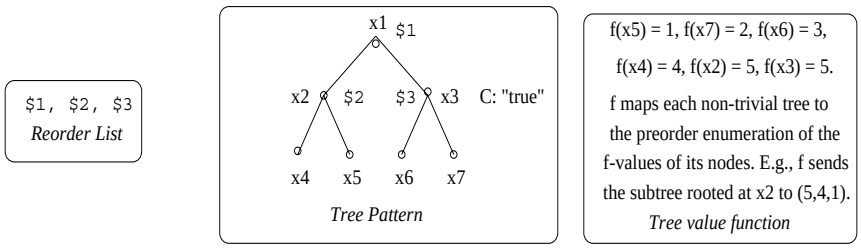

Figure 5: A counterexample showing why top-down semantics for reordering makes it non-idempotent:Tree value function specified for in.

Proof: Base case is a subtree of height two, and the lemma hold trivially. For the inductive case, consider a subtree of height $n+1$, and suppose that the lemma holds for each child of the root of this subtree (each of which is of height at most n). By definition of the reorder operator, each of these child subtrees is reordered first, and then the root is finally reordered. By the inductive assumption, re-application of the reorder operator to the child subtrees has no effect. In consequence, re-application at the root has no effect either. ∎

The semantics above corresponds to applying reordering in a bottom-up manner, in that descendants are affected before their ancestors are. This is not a trivial matter: as an example of why a top-down evaluation of reordering is not idempotent, consider the pattern and tree value function shown in Figure 5.

Consider a collection consisting of one tree which is isomorphic to the pattern itself. Applying the reordering operator with the parameters of Figure 5 top-down will produce a tree where the (images of) children of the root node $1 to be swapped. Subsequent application of reordering at nodes $2 and $3 will cause their children in turn to be swapped too. If the reordering operator is reinvoked, it will swap the children of the root node again, showing that it is not idempotent.

5.9 Copy-and-Paste

In addition to reordering siblings, we might wish to perform more general restructuring, altering the element-subelement relationship in a data tree. For instance, we may want to take a bibliography arranged with books as subelements of publishers, and transform this into a bibliography with publishers listed as subelements of books. We introduce here a simple operator κ, called *copy-and-paste*, which takes a collection as input and the following parameters: (i) a pattern $\mathcal{P}$, (ii) a copy list CL of node labels appearing in $\mathcal{P}$, and (iii) an update specification. The update specification says where the copy is to be "pasted"; example specifications include *AfterLastChild*(`$i`), *Before*(`$i`), etc., indicating the copy should be pasted after the last child of the node matching the node labeled $i in the pattern, or before such a node, etc.

The semantics of copy-and-paste operation $\kappa_{\mathcal{P},\mathrm{CL},\mathrm{US}(\mathcal{C})}$ are as follows. For each tree I in the input collection $\mathcal{C}$, do:

1. Match $\mathcal{P}$ against I. From each resulting witness tree, create a set of binding pairs: the left member of the pair for each node `$j` in the copy list, and the right member of the pair for each node `$i` in the update specification US.

2. Partition this set into subsets that have the same right member in each pair, and eliminate duplicates. In effect, this step partitions the set of binding pairs into subsets representing material to be pasted at the same spot.

3. For each partition set obtained in the previous step,

   (a) order the pairs in it based on the pedigree of the left member of the pair.

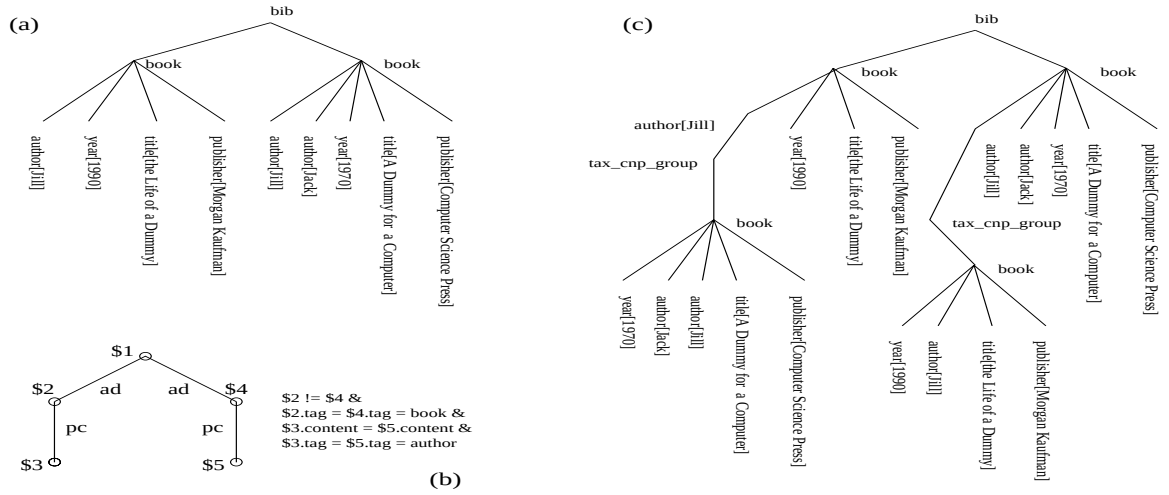

Figure 6: An example of Copy-and-paste.

(b) create a new node at the specified paste location. The tag of the new node is `tax_cnp_group` and its pedigree is null. For instance, if the paste location is *AfterLastChild*(`$i`), then the new node is placed after the current last child of the node in I that is labeled `$i`.

(c) Copy every match of `$j` in I and makes this copy a child of the node with tag `tax_cnp_group` created above. In case the copy list CL contains `$j*` in place of `$j`, then not just `$j`'s match, but the entire subtree rooted at that node, is included in the copy. Pedigrees of nodes remain unchanged in the copy. .

As an illustration, consider an invocation of copy-and-paste with the pattern tree of Figure 6(b), a copy list of `$4*`, and an update specification of *AfterLastChild*(`$3`). Applying this to the database of Figure 6(a) results in the structure of Figure 6(c). For each author of multiple books (in our case, the author Jill), we make each "other" book (along with its descendants), a descendant of each author node occurrence for this author.

5.10 Value Updates

Individual attributes of a node can be updated. We use a pattern tree to identify the nodes whose attributes need to be updated. The new value of the attribute could be a constant, the value of some other attribute in the database, or could be computed from other values (e.g., using arithmetic).

Formally, an update specification (US) is a sequence of expressions of the form `$i:attrName` $\leftarrow$ `newVal`, where `$i` is one of the node labels appearing in $\mathcal{P}$, and `attrName` is any attribute name, and `newVal` is any expression. An expression for this purpose can be either a constant, or be of one of the following forms: `$j.attr`, where `$j` is some node label in $\mathcal{P}$, or `$j.attr@attr'`, or an arithmetic expression involving the atomic expressions mentioned above. Update takes a collection $\mathcal{C}$ as input, and a pattern tree $\mathcal{P}$ and an update specification US as parameters, and generates an output collection $\upsilon_{\mathcal{P},\mathrm{US}}(\mathcal{C})$ as output, as follows.

- Every node in $\mathcal{C}$ that matches some pattern node labeled `$i` in $\mathcal{P}$, under some embedding, is marked `i`.
- For every tree in the collection $\mathcal{C}$, $\upsilon_{\mathcal{P},\mathrm{US}}(\mathcal{C})$ contains an isomorphic tree, with every attribute of every node being identical to that of the corresponding node in $\mathcal{C}$.
- Whenever a node u in the output corresponds to an input node marked `i` and the pattern node labeled `$i` in $\mathcal{P}$, and US contains the expression `$i:attrName` $\leftarrow$ `newVal`, where `newVal` is not a (string) constant, evaluate `newVal`; if it does not evaluate to a unique constant, then the update operation is undefined.

- Whenever a node u in the output corresponds to an input node marked `i` and the pattern node labeled `$i` in $\mathcal{P}$, and US contains the expression `$i:attrName` $\leftarrow$ `newVal`, then if u contains an attribute `attrName`, then its value is changed to (the unique evaluated value of) `newVal`.

Due to multiple embeddings of the pattern tree, it is possible for an input node to get marked more than once. If the update operations due to these multiple markings clash, then the update specification is ambiguous, and an implementation of extended TAX may choose to resolve the ambiguity as it sees fit.

Note the static semantics of update: first, every node that is potentially affected by the update is marked; then the `newVal` values are assigned to the RHS of the relevant expressions, and only then the appropriate changes are made in every marked node. This is needed to ensure that updating does not have a "cascade" effect, in the sense that attribute values are not changed more than once and the output is unambiguous.

5.11 Node Deletion

Deletion of nodes requires a new operator, but the basic idea still remains the same. A pattern tree is used to identify nodes, and a delete specification indicates by node label which nodes to delete. Formally, the delete operator takes a collection $\mathcal{C}$ as input, and a pattern tree $\mathcal{P}$ and a delete specification DS as parameters. A delete specification (DS) is a sequence of expressions of the form `$i` or `$i*`, where `$i` is one of the node labels appearing in $\mathcal{P}$. It generates an output collection $\delta_{\mathcal{P},\mathrm{DS}}(\mathcal{C})$ as output, as follows.

- Every node in $\mathcal{C}$ that matches some pattern node labeled `$i` in $\mathcal{P}$, under some embedding, is marked `i`.
- For every tree in the collection $\mathcal{C}$, $\delta_{\mathcal{P},\mathrm{DS}}(\mathcal{C})$ contains an isomorphic tree, with every attribute of every node being identical to that of the corresponding node in $\mathcal{C}$.
- Whenever a node u in the output corresponds to an input node marked `i` and the pattern node labeled `$i` in $\mathcal{P}$, then
  - If DS contains the expression `$i*`, then node u is deleted along with all of its descendants.
  - If DS contains the expression `$i`, then node u is deleted, and each of its children is made a direct child of u's parent. These children retain their relative order, and are inserted in the same position w.r.t. node u's siblings as node u used to be.

In short, the delete node operation is almost identical to the value update operation, except for the extra care necessary when deleting non-leaf nodes. We additionally permit delete specifications to include expressions of the form `$i.attrName`, and interpret this to mean deletion of the indicated attribute rather than deletion of the entire node.

Node deletion is very similar to projection. In fact, it can be viewed as projection with a complemented projection list, specifying nodes to be eliminated rather than nodes to be retained. Since each element in either form of list is specified by means of a pattern tree, there is no simple way in general (without introducing negation into the definition of a pattern tree and its witnesses) to transform a positive projection (retention) list into a complemented projection (deletion) list, and vice versa.

5.12 Node Insertion

When inserting nodes (rather than just attributes), we have a little more information to provide. The insert specification has to specify the position w.r.t. the identified pattern tree nodes where the insertion should be performed. Also, the insertion data has to be "marshalled". Insertion is always of a leaf node. If other nodes are to be made children of it, one can use copy-and-paste for this purpose.

Formally, the insert operator takes a collection $\mathcal{C}$ as input, and a pattern tree $\mathcal{P}$ and an insert specification IS as parameters. An insert specification (IS) is a sequence of expressions of the form `$i.attrName=val` or $<$

pos >`$i(attrName1=val1, attrname2=val2, ...)` where `$i` is one of the node labels appearing in $\mathcal{P}$, `attrName*` are names of attributes, `val*` are expressions, and < *pos* > is a relative position expression such as $AfterLastChild$ or $NextSibling$. It generates an output collection $\iota_{\mathcal{P},\text{IS}}(\mathcal{C})$ as output, as follows.

- Every node in $\mathcal{C}$ that matches some pattern node labeled `$i` in $\mathcal{P}$, under some embedding, is marked `i`.
- For every tree in the collection $\mathcal{C}$, $\upsilon_{\mathcal{P},\text{IS}}(\mathcal{C})$ contains an isomorphic tree, with every attribute of every node being identical to that of the corresponding node in $\mathcal{C}$.
- Whenever a node u in the output corresponds to an input node marked `i` and the pattern node labeled `$i` in $\mathcal{P}$, and IS contains an expression involving `$i`, evaluate each `val*` in the expression to reduce it to a constant.
- Whenever a node u in the output corresponds to an input node marked `i` and the pattern node labeled `$i` in $\mathcal{P}$,
  - If IS contains the expression `$i.attrName=val`, then node u has a new attribute added with the specified name and value. If the attribute already exists with the same name, then the attempt to insert the new attribute causes an XML error.
  - If IS contains the expression < *pos* >`$i...`, then a new node v is inserted, with the specified attributes and values, in a position relative to u specified by < *pos* >.

More general updates than the ones above are possible. In particular, it is possible to include references to old values of attributes/content in defining the new values. Once more, there is a clear notion of old and new, so no confusion arises.

Old references should only be to values and not directly to structural information. Updates involving structure are performed through cnp etc. Of course new aggregate value nodes can always be created with relevant extracts of structural information, and then these values used in any desired updates. Thus, for instance, it is possible to add an attribute to each book in some bibliography with a value equal to the total number of books in the bibliography.

6 Expressive Power of TAX

In this section, we establish results on the expressive power of TAX. First, we show that it is complete for relational algebra extended with aggregation. A central motivation in designing TAX is to use it as a basis for efficient implementation of high level XML query languages. Later in this section we examine the expressive power of TAX w.r.t. popular XML query languages.

6.1 Translating Relational Queries

Lemma 6.1 (Independence) : The operators in TAX are independent, i.e., no operator can be expressed using the remaining ones.

Proof: Follow an approach essentially similar to proof of independence of relational algebraic operators.

∎

Theorem 6.1 (Completeness for RA with Aggregation) : There is an encoding scheme Rep that maps relational databases to data tree representations such that, for every relational database D, and for every expression Q in relational algebra extended with aggregation, there is a corresponding expression Q' in TAX such that $Q'(Rep(D)) = Rep(Q(D))$. ∎

Proof: We can represent tuples as (two-level) data trees: a root node for the tuple with one child for each attribute in the tuple (see Figure 7a). Then we can show that each relational algebra operator can be simulated using TAX.

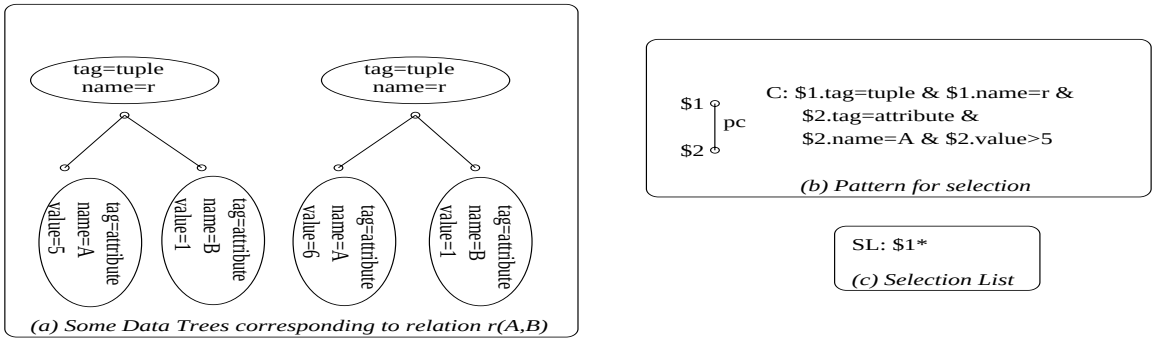

Figure 7: Simulating relational selection.

Selection on a relation r can be expressed using a pattern $\mathcal{P} = (T, C)$ that shows the complete structure of the relation's tuple data tree representation and including the selection condition as part of the formula C. One could equivalently include only the tuple root and the referenced attributes in the pattern tree. In either case, the selection list is the tuple root with all its descendants (children attributes). E.g., $\sigma_{A>5}(r)$ where $r(A, B)$ is a binary relation can be expressed using the pattern and selection list of Figure 7. Similarly, projection is simulated by means of a pattern tree that identifies the tuple root and all (or only the relevant) attribute children, and a projection list that comprises the tuple root and the set of attributes to be retained in the projection. except the predicate $\$3.\text{value} > 5$ is dropped from the formula C. Union, difference, and renaming are all straightforward. Product in TAX introduces a new root node, and would have three-level trees in the output. One must project out the second level nodes in these trees (corresponding to the roots of the trees input to the product) to get the desired relational product result. Thus we have shown that all relational algebra operators can be simulated in TAX.

Turning now to aggregation, the generic statement to consider is of the form: $\mathsf{GROUPBY}_{G_1,...,G_k;agg_1(V_1),...,agg_m(V_m)}(E)$. Let E_{tax} be the TAX expression equivalent to E. Then the required TAX expression is obtained by applying a GROUPBY to the TAX expression E_{tax} equivalent to E. The grouping basis is a list specified by depicting a pattern showing the full structure of trees in the collection returned by E_{tax} (or at least relevant components thereof) and including node labels corresponding to the tags $G_1, ..., G_k$ in the grouping list. The ordering function is defined arbitrarily, e.g., based on pedigrees. Then we apply the aggregation operator to the resulting collection. The pattern tree associated with the aggregation operator is obtained by depicting the structure of the collection resulting from grouping. The aggregate functions $agg_i(V_i)$ are included as parameters to the aggregation operator. All of them are added after the last child of the root.[5] ∎

6.2 Translating XML Queries

In this subsection, we discuss the translation of XQuery into TAX. We begin with simple single-block FLWR expressions, and gradually add features such as nesting, quantifiers, and so on, until we include most (but not all) of XQuery. For this purpose, we introduce the following definitions:

Definition 6.1 (Canonical XQuery Statement) A *canonical* XQuery statement is of the form:

```
FOR var11 IN range1,
...
    var1m in rangem
LET letvar11 := letexpr11,
...
    letvar1n := letexpr1n
```

[5] Strictly speaking, we should apply the aggregation one by one, for each aggregate function in turn. We have abbreviated this cascade as a single application of aggregation operator with many aggregate functions as parameters.

```
WHERE conditions1
RETURN
<tag1>
   <tag11>
    var11
   </tag11>
     ...
   <aggtag1i>
     agg(letvar1i)
   </aggtag1i>
     ...
...

      FOR vark1 IN rangek1,
...
          varkl IN rangekl
      LET letvarkl := letexprkl,
...
          letvarkp := letexprkp
      WHERE conditionsk
      RETURN
      <tagk>
        <tagk1>
         vark1
        </tagk1>
          ...
        <aggtagkj>
         agg(letvarkj)
        </aggtagkj>
          ...
        </tagk>
</tag1>
```

Where,

- variables bound in the LET clause are bound to aggregate expressions,
- each `conditionsi` in the WHERE clause is a quantifier-free predicate,
- all regular path expressions used involve only constants, wildcards and may further use '/' and '//', and
- there are no functional calls, tag variables, or recursion in any of the expressions mentioned.

Definition 6.2 (Single Block Canonical XQuery Statement) A *single block canonical Xquery statement* is a canonical Xquery statement (a "FLWR" expression) with no nesting.

Theorem 6.2 (Single Block Canonical XQuery Translation) : Let Q be a single block canonical XQuery statement. There is an expression E in TAX that is equivalent to Q.

Proof:
This query can be translated into TAX as follows.

Step 1 (identify all tree patterns):

- two variables $x and $y declared within a single FOR clause are related, denoted $x ≡ $y, if one of them occurs in the range of another or if they are both related to a third variable; clearly, ≡ is an equivalence relation; split variables declared within each FOR clause into equivalence classes based on ≡;
- for each equivalence class associated with each FOR clause, construct a pattern tree based on the declarations as follows.
  * Create one node corresponding to each variable, and one edge from $x to $y whenever $y occurs in the range of $x's declaration.
  * If any node has in-degree greater than one, split it into multiple nodes of in-degree one, replicating the sub-tree below the node.
  * If the same variable name corresponds to multiple nodes, create predicates equating the nodes.
  * Expand each edge into an appropriate sequence of ad and pc edges in the pattern tree, creating intermediate nodes as required, based on the (partial) path expression corresponding to the edge.
  * Instantiate node predicates corresponding to appropriate nodes from the path expression(s).

Step 2 (push conditions to patterns): analyze the WHERE clauses and push any conditions involving only variables from a single pattern tree to the relevant pattern tree predicate.

Step 3 (duplicate elimination): whenever the DISTINCT keyword is used, perform the corresponding duplicate elimination, as described in Section 5.5.

Step 4 (evaluate LET aggregates): use an aggregation operator, with an appropriate grouping, to evaluate aggregate expressions in LET clauses.

Step 5 (form the joins): compute the join of multiple pattern trees, if any, obtained from the previous steps. With each join, associate as join conditions any predicates from the WHERE clause that reference variables in exactly the trees being joined.

Step 6 (remaining conditions in WHERE clauses): enforce any remaining conditions in the WHERE clauses

Step 7 (evaluate RETURN aggregates): use an aggregation operator, with an appropriate grouping, to evaluate aggregate expressions in RETURN clauses.

Step 8 (ordering): perform a groupby, using the elements RETURNed for defining the grouping basis, and the ordering tree value function defined based on the ORDER BY clauses if any; the default is pedigree ordering.

Step 9 (projection): Based on the RETURN statement arguments, form the projection list, using copy-and-paste as necessary.

The resulting TAX expression E is equivalent to the query Q. ∎

The above result applied to a single block XQuery statement. Nested blocks in the WHERE or RETURN clauses can be handled quite easily. Leading to the following result:

Theorem 6.3 (Canonical XQuery Translation) : Let Q be a canonical XQuery statement. There is an expression E in TAX that is equivalent to Q.

Proof: Use the same translation procedure as before, except that in Step 5, we arrange the various identified pattern trees in the nesting sequence, inside out, and then form a cascade of left-outer-joins; e.g., if we have patterns $T1$, $T2$, $T3$, we form $((T1 \bar{\bowtie} T2) \bar{\bowtie} T3)$; Once again, with each join, associate as join conditions any predicates from the WHERE clause that reference variables in exactly the trees being joined. Also, in Step 8, instead of just a single grouping to order outputs, a cascade of grouping operators are applied in the nesting sequence, inside out. ∎

We now consider some XQuery facilities not included in a canonical XQuery statement. XQuery permits the same variable to be declared over multiple ranges – either explicitly or implcitily via INTERSECT. An example of explicit declaration is FOR $a IN //book/author, $a IN //university//faculty, ... while an implicit declaration is FOR $a IN (//book/author INTERSECT //university//faculty). Note that FOR $a IN (//book/author EXCEPT //university//faculty) is *not* an example of declaring a variable over multiple ranges.

Lemma 6.2 (Eliminating Multiple Range Declarations) : For every XQuery statement there is an equivalent XQuery statement in which no variable is declared over multiple ranges.

Proof: Let Q be a XQuery statement and let $x be a variable declared over the ranges range1, ..., rangek. Let the declarations be FOR $x IN (range1 INTERSECT ... INTERSECT rangek). Then replace this declaration by the declarations FOR $x1 IN range1, ..., $xk IN rangek. Add the condition $x1 = $x2 AND ... $x1 = $xk to the WHERE clause associated with this FOR clause. Replace every occurrence of $x in the associated RETURN statement by any $xi. It is easy to see that this transformation preserves the equivalence of queries. A repeated application of this transformation yields the desired equivalent query. ∎

Lemma 6.3 (Quantifiers) : Every XQuery statement involving quantifiers in WHERE clauses can be rewritten into an equivalent quantifier free statement in XQuery extended with a set difference facility.

Proof Sketch: Any occurrence of SOME quantifiers can be easily rewritten by:

1. dropping the quantifier expression from the WHERE clause,
2. declaring the quantified variable in the immediately surrounding FOR clause with the same range over which it was originally quantified, and
3. adding the condition following SATISFIES to the WHERE clause where the quantifier originally occurred.

This transformation clearly preserves equivalence. By repeating this transformation for every occurrence of the SOME quantifier, we can generate an equivalent XQuery statement that does not use this quantifier.

Next, consider an occurrence of the EVERY quantifier. Let the quantified expression be EVERY $x IN (<range>) SATISFIES cond, occurring some query Q. Create query $Q1$ to be identical to Q, except that this quantified expression is deleted. Create query $Q2$ to be identical to Q except that this quantified expression is replaced by the expression SOME $x IN (<range>) SATISFIES (NOT cond). Clearly, the result of Q can be obtained as the result of $Q1$ minus the result of $Q2$. Also, each of $Q1$ and $Q2$ has one less EVERY quantifier than Q. (The SOME quantifier in $Q2$ is shown here for ease of exposition, and is immediately removed using the technique shown above). Since there are only a finite number of quantifiers in any query, this process can be applied repeatedly until no more quantifiers remain in the query. ∎

In summary, we have shown how a canonical XQuery statement can be expressed in TAX. Then we showed how various XQuery statements could be reduced to canonical form. From here, by exhaustive consideration of XQuery constructs, one can establish the following claim:

Theorem 6.4 (Translating XQuery) : Let Q be an XQuery query not involving recursion, function calls or tag variables, which further satisfies the following:

- the variables bound in the LET clauses are bound to an aggregate expression; and
- all regular path expressions used involve only constants or wildcards and may further use '/' and '//'.

Then there is an expression E in TAX that is equivalent to Q. ∎

Note that the precise definition of XQuery is a moving target, and the claim above could be affected by the addition of powerful new constructs to XQuery. In the event that some XQuery construct is missed by us in our exhaustive consideration, or added thereafter by the W3C committee, such a construct will either be expressible in canonical form, or can be recorded as an additional exception in the theorem statement.

Similar translation theorems can be shown for Quilt, XML-QL, XQL, and so on, suppressed here for brevity.[6] From our experience, we have found that most interesting XML queries arising in practice can be translated to TAX. We have consciously chosen to keep recursion outside the algebra, and have thus managed to devise a clean algebra with a small set of simple and intuitive operators. Furthermore, for queries involving recursion, an implementation of TAX can provide an explicit support for iteration. This is similar to implementing deductive databases via relational algebra plus iteration.

[6] Interestingly, several queries requiring the use of Skolem functions in XML-QL can be expressed in TAX, which doesn't have this feature.

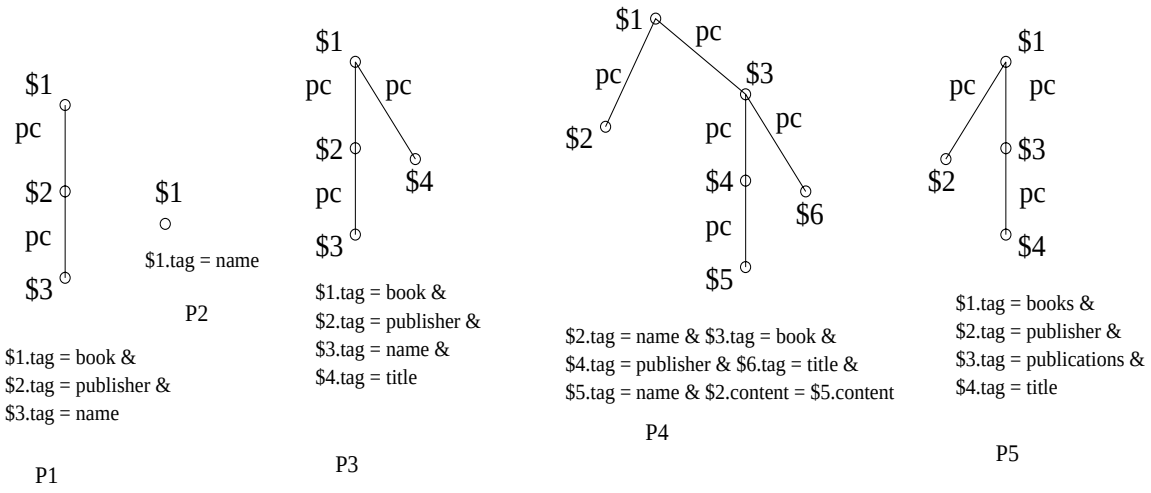

Figure 8: Pattern trees for translation of a XQuery query (Ex:6.2) into TAX

6.3 XQuery Examples

Example 6.1 Consider a simple XQuery query.

```
<Result>
FOR       $b IN document("http://www.biblio.com/books.xml")//book,
          $a IN $b/author
WHERE     $a/firstname = "Mark" AND $a/lastname = "Twain"
RETURN    $b
</Result>
```

This query can be expressed in TAX with a straightforward selection.[7] ∎

Example 6.2 Consider the classic XQuery query that takes a document arranged by book, with publisher a subelement of book, and rearrange it by publisher, ordering books under each publisher by title lexicographically:

```
FOR $p IN distinct(document("x.xml")//book/publisher/name)
RETURN
     <books>
          <publisher>
               <name> $p </name>
          </publisher>
          <publications>
               FOR $b IN document("x.xml")//book[publisher/name = $p],
                    $t IN $b/title
               RETURN
                    <title> $t </title>
               ORDERBY $t
          </publications>
     </books>
```

[7] Note that the output is constructed as a single (ordered) tree, not as a collection. This can be easily accomplished in TAX by grouping the collection by nothing, and ordering the trees in the collection by the term representation of the pedigrees in each tree. Since this is a common idiom, we will not repeat this step in future examples.

A straightforward translation of this yields:

$E_0 = DE(\pi_{P2,\ \{\$1\}}(\sigma_{P1,\{\}}(\mathcal{C})))$.

Here $P1$, $P2$ etc. are pattern trees defined in Figure 8, $\mathcal{C}$ is the input collection, and E_0 is an intermediate result comprising the collection of bindings for the publisher name variable, `$p`. DE is shorthand for duplicate elimination by value, obtained as a grouping (by everything) operator followed by projection (see Lemma 5.2). We then form:

$E_1 = E_0 \ \bar{\bowtie}_{P4} \ (\sigma_{P3,\{\}}(\mathcal{C}))$.

Here, $\bar{\bowtie}$ denotes left outerjoin (taking the pattern $P4$ as parameter). E_1 is another intermediate result after evaluating the inner FOR loops and constraining variable values through a left outerjoin on `$p`. Next, we obtain:

$E_2 = \gamma_{P3,\$3.\mathtt{content},\$4.\mathtt{content},\mathtt{books},\mathtt{publications}}(E_1)$.

E_2 puts the output together, in correct structure and order. Finally, the desired output is produced by a simple projection:

$\pi_{P5,\{\$1,\$2*,\$3,\$4\}}(E_2)$.

Example 6.3 The next query illustrates the LET and FILTER features of XQuery. The input document is a cookbook (cookbook.xml) containing sections nested to arbitrary depth and containing section title as well as figures which may themselves contain their titles. The following query filters out sections and their titles, while preserving the original hierarchical structure.

```
<toc> (
          LET $s := document("cookbook.xml")
                    FILTER //Section | //Section/Title)
          RETURN $s
)
</toc>
```

This can be expressed as a simple projection in TAX, with a two-node pattern tree which says the root tag is Section and the child tag is Title. The sole edge in the pattern is a pc-edge. ∎

Example 6.4 Consider a query that involves selection based on aggregate values.

```
FOR       $pub IN DISTINCT //publisher
LET       $bk := //book[pubinfo/publisher=$pub AND pubinfo/year=``1999"]
WHERE     count($bk) > 100
RETURN $pub
```

This query can be expressed in TAX as shown in Figure 9, where we only show the initial selection choosing books published in 1999 and the aggregation counting such books published by each publisher. Subsequently, we need to select those publishers with count > 100 and join the result with the set of (distinct) publishers out there. The latter steps are routine. ∎

Example 6.5 The next query illustrates how outerjoins are expressed. Consider an auction database consisting of the tables *users(userid, name, rating), items(itemno, description, offeredby, reserveprice), bids(userid, itemno, amount, date)*. The following XQuery query, which assumes suitable XML representation of these tables, computes the left outerjoin of `users` and `items`.

```
<result>
(
```

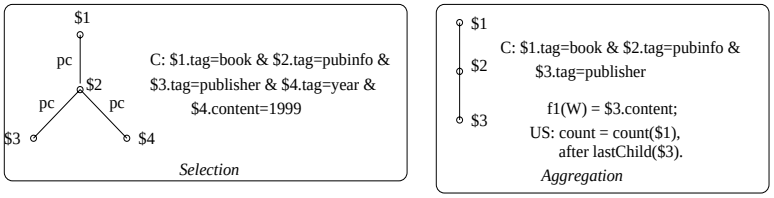

Figure 9: Expressing XQuery query with aggregation.

```
FOR $u IN document("users.xml")//usertuple
RETURN
          <user>
          $u/name,
          (
          FOR $i IN document("items.xml")//itemtuple
                 [itemno = $u/offeredby]
                 RETURN
                 <offering>
                 $i/description/text()
                 </offering> SORTBY(.)
          )
          </user> SORTBY(name)
)
</result>
```

∎

Example 6.6 A last example query, also drawn directly from [7], concerns surgical data where data is embedded in free text. The following query finds those sections entitled "Procedure" where no anesthesia occurs before the first incision.

```
FOR $proc IN //section[section.title="Procedure"]
WHERE NOT exists( $proc//Anesthesia BEFORE ($proc//Incision)[1] )
RETURN $proc
```

This query can be expressed in TAX as follows. First, project the input on a pattern with a section root with title "Procedure" which has an incision descendant. Next, select for each procedure its first incision child in the previous result. Third, join this with the set of procedure elements (obtained via simple selection) by equating the pedigree of the first incision child above with that of an incision descendant of the procedure element, that has an anesthesia element preceding it (using *before* predicate in pattern formula). Then project the result of the join on the procedure elements. Finally, subtract this from the set of all procedure elements. The tax query is shown in Figure 10. ∎

6.4 Translating other XML Query Languages

TAX is not specifically designed for XQuery. In fact, we have translation theorems along the lines of Theorem 6.4 for other XML Query languages, such as those discussed in [5, 17]. We omit details for lack of space, presenting instead just one illustrative example for XML-QL.

Example 6.7 The following query pulls out from a bibliography database, journal papers and books. Depending on the type, it also retrieves the publication month or the publisher.

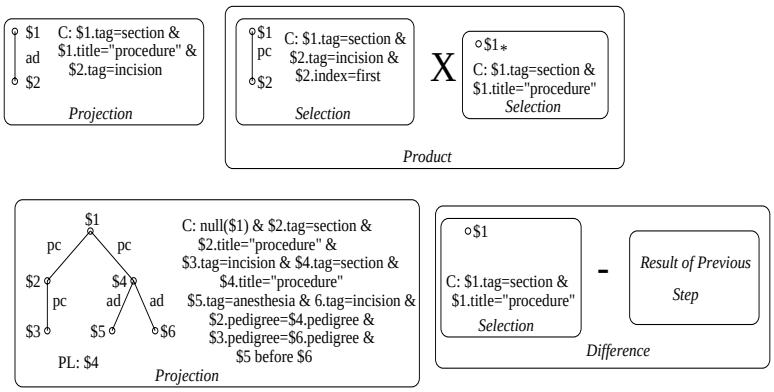

Figure 10: Expressing order-based XQuery query.

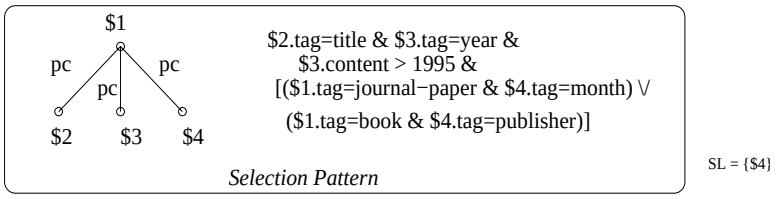

Figure 11: Expressing XML-QL queries with Skolem functions via disjunction.

```
WHERE     <$e> <title> $t </>
               <year> 1995 </> </> CONTENT_AS $p IN "www.a.b.c/bib.xml"
CONSTRUCT <result ID=ResultID(p)><title>t </> </>
          { WHERE $e = "journal-paper",
               <month> $m </> IN $p
          CONSTRUCT <result ID=ResultID($p)> <month> $m </> </> }
          { WHERE $e = "book",
               <publisher>$q </> IN $p
          CONSTRUCT <result ID=ResultID($p)> <publisher>$q </> </> }
```

This can be expressed with a simple selection in TAX (see Figure 11), while XML-QL uses heavyweight features like Skolem functions and tag variables to express this query. Note the disjunction in the formula F of the pattern. The obvious renaming step that should follow selection is omitted. ∎

6.5 New Expressions

We have shown above how large parts of XML-QL and XQuery can be expressed in terms of TAX. Furthermore, these expressions are small in size, and in some cases, simpler even than the originals. Here we present TAX queries that cannot be expressed efficiently in XMLQL or XQuery.

Example 6.8 Suppose we have two collections – of books and of reviews, as in Figure 3. Suppose we want to take a join of these collections in the sense that we want to match whichever elements have common tags between these two collections. Since there could be any number of common tagged pairs of elements across a pair of trees (e.g., title, year, etc.). We wish to include in the join only pairs where there is no mismatch.

We begin by taking a Cartesian product of the two collections. Then we remove from this product any items that satisfy the condition

`isRoot($1) && $5.tag=book && $2.tag=review && $4.tag=$7.tag && $4.content≠$7.content`

This is a very high level way of expressing a non-trivial query. Expressing this query in XQuery or XML-QL is hard because we will need to specify inclusion of exactly those pairs for which all defined subelements match, but having some subelements missing is alright. ∎

7 Optimization and Evaluation

7.1 Implementation Issues

In a typical relational query implementation, the first step is a selection, based on an index if available, or else through a full scan. Joins are implemented on the data streams that result. A similar strategy has been adopted in the TIMBER implementation of TAX. The first thing that happens is the matching of a pattern tree, which could be through a database scan, an indexed access, or a combination of indexed accesses and targeted processing of index entries. (See [3] for a study of alternative access methods for pattern tree matching.) Once witness trees (embeddings of the pattern tree in the database) have been found, each operator manipulates these witness trees as required. Note that pattern trees are independently specified by each operator in a TAX expression. The first, typically a selection, operator actually finds witnesses in the base data. Subsequent operators evaluate pattern tree embeddings on suitable intermediate results.

Finally, a word about pedigree. TAX assumes the availability of pedigree — where would this come from in a real system? There is no unique answer, but the TIMBER system uses the position of an element in a document for this purpose. In fact, the introduction of this additional attribute is costless, because it is required as part of the physical implementation, serving a role akin to `RId` in a relational database. A very similar notion is used in the Niagara system [29], bearing testament to the "naturalness" of this notion.

7.2 Derived Operators

Join, one of the most important operators in relational database implementations, is regarded a derived operator. Yet, in terms of expressing queries as well as of evaluating them, one thinks of joins directly. Similar arguments apply in TAX. Using just the primitive TAX operators, some simple tasks could require complex expressions. Appropriate derived operators can help. Moreover, direct implementation of some of these derived operators can be substantially more efficient than evaluation of a sequence of primitive operators. We have seen the value of join and left-outer-join operators above. Other derived operators can be defined as needed.

7.3 Operator Identities

Operator identities are essential to query rewriting and optimization. Tax operators have most of the usual identities one would expect. For instance, set union and intersection are associative and commutative; all TAX operators except Groupby (subject to appropriate constraints on predicates that may appear in pattern trees) distribute over set operations; and so on. For brevity, we only discuss a few issues of particular interest.

Product is not commutative since data trees are ordered. Furthermore, it is not associative as shown in Figure 12. However, Cartesian product immediately followed by a reorder on the root node of the result is indeed commutative. Similarly, Cartesian product can be rendered associative by projecting out the virtual product root node due to the first product operation, which is now a child of the root after the second product operation. This extra node is retaining information regarding the parenthesization that we wish to lose to assure associativity. Once this node is projected out, the result is a single product root with three symmetric children, thereby assuring the associativity of the product. Similar observations hold for joins.

The associativity and commutativity of join is critical for the join reordering central to much of query optimization. In light of the foregoing discussion, join operations can be reordered in a TAX query optimizer, provided that enough care is exercised. Specifically, TAX expressions can have the additional reorder and project operators inserted where

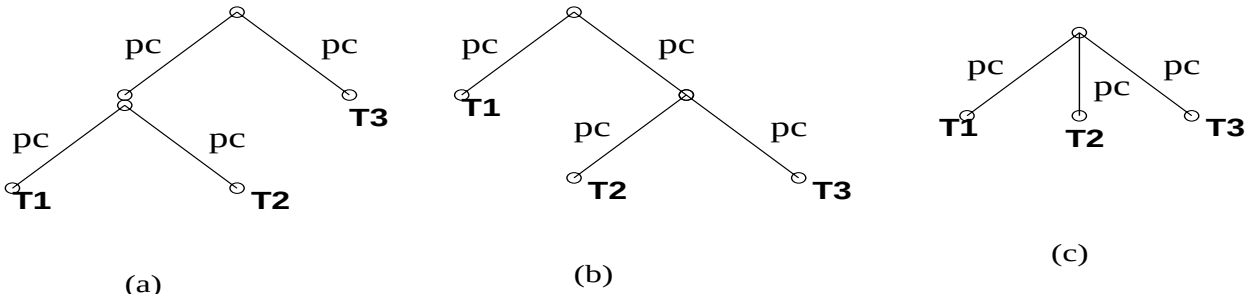

Figure 12: (a) $(\mathcal{C}_1 \times \mathcal{C}_2) \times \mathcal{C}_3$ (b) $\mathcal{C}_1 \times (\mathcal{C}_2 \times \mathcal{C}_3)$, and (c) An associative product obtained after projection

such insertions can be shown not to affect the final answer, and then these operators can be combined with the joins to render them associative/commutative as shown above. There are many cases where the final answer will not be affected — examples include when the result of a join is projected on to one of the two operands of the join, for self-joins, and so on. Besides, the associative version of join seems to be more natural in practice.

8 Related Work

There is no shortage of algebras for data manipulation. Ever since Codd's seminal paper [9] there have been efforts to extend relational algebra in one direction or another. Klug's work on aggregation [25] is worth mentioning in particular, as is the stream of work on the nested relational model. There is a grammar-based algebra for manipulating tree-structured data [20], shown equivalent to a calculus. The tree manipulations are all performed in the manner of production rules, and there is no clear path to efficient *set-oriented* implementation. Also, this work predates XML by quite a bit, and there is no obvious means for mapping XML into this data model.

Tree pattern matching is a well-studied problem, with notions of regular expressions, grammars, etc. being extended from strings to trees (*cf.* [14, 21]). These ideas have been incorporated into an object-oriented database, and an algebra developed for these in the Aqua project [30]. The focus of this algebra is the identification of pattern matches, and their rewriting, in the style of grammar production rules. Our notion of tree pattern and witness trees follows Aqua in spirit. However, Aqua has no counterpart for most TAX operators.

In the context of the Web, we should mention GraphLog [12], Hy+ [11], etc., and the recently proposed models for semi-structured data (see, e.g., Lorel [1] and UnQL [6]); all propose query languages, with more or less effort at an accompanying algebra.

Even in the XML context, several algebras have been proposed. [4] is an influential early work that has impacted XML schema specification. However, there is no real manipulation algebra described in that paper. [16] proposes an algebra carefully tailor-made for Quilt. This algebra, like the XDuce system [22], is focused on type system issues. It forms the intellectual basis for the recently issued XQuery algebra document [15] by the W3C working group on XML Query. While the algebra would be useful in investigating the semantics of XML Query, in detecting errors, and in proving query programs correct, it appears unlikely that this algebra will in itself form the basis of an effective implementation. In [10], the authors present an algebra for XML, defined as an extension to relational algebra, that is practical and implemented. However, the main object of manipulation in this algebra, as in XML-QL, is the tuple and not the tree. A "bind" operator is used to create sets of (tuples of) bindings for specified labeled nodes. Due to the consequent loss of structure, this scheme very quickly breaks down when complex analyses are required. Similarly, [26] describes a navigational algebra for querying XML, treating individual nodes as the unit of manipulation, rather than whole trees. Finally, [31] deals with many aspects of XML updates. In this paper, we do not consider updates.

9 Summary and Status

We have presented TAX, a Tree Algebra for XML, which extends relational algebra by considering collections of ordered labeled trees instead of relations as the basic unit of manipulation. In spite of the potentially complex structure of the trees involved, and the heterogeneity in a collection, TAX has only a couple of operators more than relational algebra. Furthermore, each of its operators uses the same basic structure for its parameters.

One potential complication in XML manipulation is the existence of order within a tree. Since collections are unordered, it is not straightforward to "maintain document order" in output results, for example. TAX is able to handle such requirements efficiently. In fact, the comfortable melding of ordered and unordered artifacts is one of the key intellectual contributions of TAX.

While we believe that the definition of TAX is a significant intellectual accomplishment, our primary purpose in defining it is to use it as the basis for query evaluation and optimization. We are currently building the TIMBER XML database system using TAX at its core for query evaluation and optimization. Work on query optimization is currently underway.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal on Digital Libraries*, 1(1), 1996.

[2] M. Akinde, D. Chatziantoniou, T. Johnson, and S. Kim. The MD-Join: An operator for complex OLAP. *Proc. of the International Conference on Data Engineering*, 2001.

[3] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Twig Joins: Efficient matching of XML query patterns. Submitted for publication.

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

[5] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *ACM SIGMOD Record* Volume 29, Number 1, March 2000, pp. 68-79.

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

[7] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proc. Int. Workshop on Web and Databases*, May 2000.

[8] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A query language for XML. W3C Working Draft. 15 Feb. 2001.

[9] E. F. Codd. A relational model of data for large shared data banks. *CACM* 13(6), pp. 377–387, 1970.

[10] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. SIGMOD*, pages 141–152, 2000.

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

[12] M. P. Consens and A. O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proc. PODS*, Apr. 1990.

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proc. Int. World Wide Web Conf*, 1999.

[14] J. Doner. Tree acceptors and some of their applications *JCSS* Vol. 4, pages 406-4451, 1970.

[15] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. The XML Query Algebra. W3C Working Draft. 15 Feb. 2001.

[16] M. Fernandez, J. Simeon, and P. Wadler. An algebra for XML query. In *Proc. FST TCS*, Delhi, December 2000.

[17] M. Fernandez, J. Simeon, and P. Wadler (editors). XML Query Languages: Experiences and Examplars, draft manuscript, communication to the XML Query W3C Working Group, September 1999.

[18] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. IEEE Data Eng. Bulletin 22(3): 27-34 (1999).

[19] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB*, 1997.

[20] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. In *Proc. ACM SIGMOD*, pages 263–272, 1989.

[21] C. M. Hoffmann and M. J. O'Donnell. Pattern-matching in trees. *JACM* Vol. 29, pages 68–95, 1982.

[22] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *Proc. Int. Workshop on Web and Databases*, May 2000.

[23] H. V. Jagadish, Laks V. S. Lakshmanan, Tova Milo, Divesh Srivastava, and Dimitra Vista. Querying Network Directories. In *Proc. ACM SIGMOD*, Philadelphia, PA, June 1999.

[24] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD*, pages 393–402, 1992.

[25] A. C. Klug. Calculating constraints on relational expressions. *TODS* 5(3) pp. 260–290, 1980.

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

[27] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. PODS*, 2000.

[28] J. Robie (ed.). XQL '99 proposal. `http://metalab.unc.edu/xql/xql-proposal.html`

[29] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. VLDB*, 1999.

[30] B. Subramanian, T. W. Leung, S. L. Vandenberg, S. B. Zdonik. The AQUA approach to querying lists and trees in object-oriented databases. In *Proc. ICDE*, 1995.

[31] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. SIGMOD*, 2001.

[32] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[33] World Wide Web Consortium. The document object model. `http://www.w3.org/DOM/`

[34] WWW Consortium. XML Query Language Working Group Documents. `http://www.w3.org/XML/Query.html`

[35] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. SIGMOD*, 2001.