TAX: A Tree Algebra for XML

H. V. Jagadish¹, Laks V. S. Lakshmanan², Divesh Srivastava³, and Keith Thompson¹

¹ University of Michigan, Ann Arbor, MI 48109, USA
² University of British Columbia, Vancouver, BC V6T 1Z4, Canada
³ AT&T Labs–Research, Florham Park, NJ 07932, USA
jag@eecs.umich.edu, laks@cs.ubc.ca, divesh@research.att.com, 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 [7], Quilt [6], and XML-QL [11] have recently been proposed for this purpose.

This leads us to the question of implementation. If we expect to have large XML data sets, 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 [22] 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 [17]. 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, rooted 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 summarize the expressive power of TAX. We discuss related work in Section 7 and conclude in Section 8.

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, but not its subelements) 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, 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 [23], 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 [11], 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 opera-
tions, 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, and hence 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 few additional operators to deal with manipulation of tree structure.

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, rooted tree, the *data 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, the information content
in the node represents the attributes of the element, while its children nodes
represent its subelements. For XML, we assume each node has a special 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. 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. As we shall show later, appropriate use of the pedigree 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 rather than sets. Similarly, 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.

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, and this can have a significant effect on the complexity of expression evaluation. For concreteness, we use a representative set, listed below, with a clear understanding that this set is extensible.

---

1 Pedigrees are not shown in our example data trees to minimize clutter.
For a node (element) $i$, any attribute $attr$ and value $val$ from its domain, the atom \( i.attr \theta val \) is allowed, where $\theta$ is one of $=, \neq, >$, etc.\(^2\) As a special case, when $attr$ is of type string, a wildcard comparison such as $i.attr = "*val*"$, where $val$ is a string, is allowed. Similarly, for two nodes $i$ and $j$, and attributes $attr$ and $attr'$, the atom $i.attr \theta j.attr'$ is allowed. Specifically, the attribute could be the pedigree: predicates of the form $i.pedigree \theta j.pedigree$, where $\theta$ 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, $i.index = 1$ means that node $i$ is the first child of its parent. More generally, $\text{index}(i, j) = n$ means that node $i$ is the $n^{th}$ node among the descendants of node $j$. Similarly, $i \theta j$, where $\theta$ is one of $=, \neq, \text{BEFORE}$, means that node $i$ is the same as, is different from, or occurs before node $j$. These positional predicates are based on the preorder enumeration of the 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 names in relational algebra. However, collections of XML data trees are typically heterogeneous. Besides, 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 1 (Pattern Tree)** Formally, a pattern tree is a pair \( P = (T, F) \), where $T = (V, E)$ is a node-labeled and edge-labeled tree such that:

- each node in $V$ has a distinct integer (denoted $i$) as its label;
- each edge is labeled $pc$ (for parent-child) or $ad$ (for ancestor-descendant),
- $F$ is a formula, i.e., a boolean combination of node predicates.

\(^2\) We also allow the variants $i.attr = val$ (meaning val appears as a value of some attribute) and $i.attr \neq \_$(meaning attribute attr is defined for node $i$).
While the formal semantics of patterns are given in the next subsection, here we give some examples. Figure 1(b) shows a pattern that matches books published before 1988 and having at least one author. The edge label \texttt{pc} indicates that \texttt{year} must be a direct subelement of \texttt{book}, while the edge label \texttt{ad} indicates that \texttt{author} could be any nested descendant subelement. As another example, the pattern in Figure 1(c) matches books published by a publisher 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 (\texttt{ad}) edges, in addition to the basic parent-child (\texttt{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.

Pattern trees in TAX also permit attributes of nodes to be compared with other node attributes, analogously to selection predicates in relational algebra permitting different attributes to be compared. See, for instance, Figure 1(c), where the positions of two nodes are compared using the \texttt{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 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 \texttt{pc} (resp., \texttt{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 $n$ a node in $\mathcal{C}$ such that $n = h(u)$. Then we say the data tree node $n$ 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 use of ordering predicates (such as $\text{BEFORE}$), 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. Thus, $\text{TAX}$ permits the graceful melding, even within a single query, of places where order is important and places where it is not.

We next formalize the semantics of pattern trees using a notion of witness trees induced by embeddings of a pattern tree into a database:

**Definition 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 $h^C(\mathcal{P})$ 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 $(m, n)$. Intuitively, each edge in the witness tree corresponds to a sequence of one or more edges in the input data tree that has been collapsed because only the end-point nodes of the sequence are retained in the witness tree.
- the witness tree preserves the order on the nodes it retains from $\mathcal{C}$, i.e., for any two nodes in $h^C(\mathcal{P})$, whenever $m$ precedes $n$ in the preorder node enumeration of $\mathcal{C}$, $m$ precedes $n$ in the preorder node enumeration of $h^C(\mathcal{P})$. 

![Figure 2. Results of various operations applied to the database of Figure 1(a)](https://example.com/fig2.png)
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 $h^c(\mathcal{P})$. We also refer to $h^c(\mathcal{P})$ 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 places, then multiple witness trees are obtained, one for each embedding. For example, Figure 2(a) shows three witness trees resulting from embedding the pattern of Figure 1(b) into the database of Figure 1(a) in three different places. The structure of all three witness trees is the same three-node structure, by definition, but the database nodes bound are different. It is permissible for the same database node to appear in multiple witness trees. For instance, the same book appears in the first and second 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). While the exact nature of TVFs may be orthogonal to the algebra, we assume below they are primitive recursive functions on the structure of their argument trees. We typically assume the codomain of a TVF is (partially) ordered. When used for sorting purposes, it must be totally ordered. 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 5.4. Just like pattern trees, TVFs are used in conjunction with a variety of operators in our algebra.

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 book collection. 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 would return the
original data tree, leaving no clue about which book was published before 1988.

Selection in TAX takes a collection \( C \) as input, and a pattern \( \mathcal{P} \) and adorn-
ment \( \mathcal{S} \) 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 \( \mathcal{S} \). The adornment list, \( \mathcal{S} \), lists nodes from \( \mathcal{P} \) for which
not just the nodes themselves, but specified structural “relatives” (e.g., siblings,
parent, ancestors, descendants, etc.) of it, need to be returned. A frequently
used important special case is the set of descendants of a node. (An element
is expected to include all nested subelements in typical XML and XQuery sem-
antics, for instance.) To keep the exposition as simple as possible, we restrict
adornments in the foregoing to all descendants: if a node is mentioned in the
adornment list, all its descendants are returned in addition to the witness tree. If
the adornment list is empty, then just the witness trees are returned. Formally,
the output \( \sigma_{\mathcal{P},\mathcal{S}}(\mathcal{C}) \) 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 \( n \) in the input collection \( \mathcal{C} \) belongs to the output iff \( n \) matches
  some pattern node in \( \mathcal{P} \) under \( h \), or \( n \) is a descendant of a node \( m \) in \( \mathcal{C} \)
  which matches some pattern node \( w \) under \( h \) and \( w \)'s label appears in the
  adornment list \( \mathcal{S} \).
- Whenever nodes \( n, m \) belong to the output such that among the nodes re-
tained in the output, \( n \) is the closest ancestor of \( m \) in the input, the output
  contains the edge \( (n, m) \). Intuitively, the output tree preserves the structure
  of the input, restricted to the retained nodes.
- The relative order among nodes in the input is preserved in the output, i.e.,
  for any two nodes \( n, m \) in the output, whenever \( n \) precedes \( m \) in the preorder
  enumeration of \( \mathcal{C} \), \( n \) precedes \( m \) 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 \( \sigma_{\mathcal{P},\mathcal{S}}(\mathcal{C}) \) produces the collection of trees in
Fig 2(a) if the adornment list \( \mathcal{S} \) is empty. On the other hand, if \( \mathcal{S} \) 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.
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 \( C \) as input and a pattern tree \( P \) and a projection list \( PL \) as parameters. A projection list is a list of node labels appearing in the pattern \( P \), possibly adorned with \( * \). The output \( \pi_{P, PL}(C) \) of the projection operator is defined as follows:

- A node \( n \) in the input collection \( C \) belongs to the output if and only if there is an embedding \( h : P \rightarrow C \) such that \( n \) matches some pattern node in \( P \) whose label appears in the projection list \( PL \), or \( n \) is a descendant\(^3\) of a node \( m \) in \( C \) which matches some pattern node \( w \), and \( w \)'s label appears in the projection list \( PL \) with a "\(^*\)".
- Whenever nodes \( n, m \) belong to the output such that among the nodes retained in the output, \( n \) is the closest ancestor of \( m \) in the input, the output contains the edge \((n, m)\). Intuitively, the output tree preserves the structure of the input data tree, with every edge in the output tree corresponding to an ancestor-descendant path in the input data tree.
- The relative order among nodes is preserved in the output, i.e., for any two nodes \( n, m \) in the output, whenever \( n \) precedes \( m \) in the preorder enumeration of \( C \), \( n \) precedes \( m \) 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 \( \{S1, S2, S3\} \), 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 add a new root node labeled \( S0 \) to the pattern tree, with an ad edge to the previous root of the pattern tree, and include \( S0 \) in the projection list \( PL \).

Projection can also be used to return entire trees from the input collection that have an embedding of a pattern tree. To do so, all we have to do is to add a new root node labeled \( S0 \) to the pattern tree, with an ad edge to the previous root of the pattern tree, and include \( S0^* \) in the projection list \( PL \).

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.

\(^3\) Other relatives are permitted as for selection, but suppressed in the exposition.
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. Compare the projection result shown in Figure 2(b) for the pattern tree of Figure 1(b) and the database of Figure 1(a), 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 $C$ and $D$ as input and produces an output collection corresponding to the “juxtaposition” of every pair of trees from $C$ and $D$. More precisely, $C \times D$ produces an output collection as follows.

- for each pair of trees $T_i \in C$ and $T_j \in D$, $C \times D$ contains a tree, whose root is a new node, with a tag name of `tax:prod_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 $T_j$.
- 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, $C \times D$ and $D \times C$ 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.

5.4 Grouping

Unlike 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. Due to lack of space, we present only grouping here.

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.

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. Formally, the groupby operator $\gamma$ takes a collection as input and the following parameters.

- A pattern tree $P$; this is the pattern used for grouping. Corresponding to each witness tree $T_j$ of $P$, we keep track of the source tree $I_j$ from which it was obtained.
– A **grouping tree value function** that partitions the set \( W \) of witness trees of \( P \) against the collection \( C \). Typically, this grouping function will be instantiated by means of a **grouping list** that lists elements (by label in \( P \)), and/or attributes of elements, whose values are used to obtain the required partition. The default comparison of element values is “shallow”, ignoring subelement 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 \( \text{orfun} \) that maps data trees to a totally ordered domain. This function is used to order members of a group for output, in the manner described below.

The output tree \( S_i \) corresponding to each group \( W_i \) is formed as follows: the root of \( S_i \) has tag **tax_group_root**, a null pedigree and two children; its left child \( \ell \) 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 \( W_i \), arranged in increasing order w.r.t. the value \( \text{orfun}(T_j) \), \( i \) 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 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. 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 group, so their existing structure is preserved. 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, the original 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 3. 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. Lastly, if we apply grouping to this same collection using a **TVF** that maps each book to its number of authors, then we will obtain a collection of books grouped by number of authors, with the books in a group ordered in a manner dictated by the ordering **TVF**.

A few words regarding the way collections of source trees are partitioned are in order. For every node label of the form \( \text{\$1} \) 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). Note the difference with tree equality, based on isomorphism that preserves pedigree as well.

In short, equality can be shallow or deep, and it can be by value (without pedigree) or by complete tree equality (including 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 that duplicate elimination of nodes by value can be expressed in TAX.

**Other Operators:** As in the relational model, we fall back on set theory for set union, intersection and difference. One can also define operators for aggregation, for renaming, and for structural manipulation of trees (e.g., reordering). Both pattern trees and TVFs play a central role in their definitions. These are discussed in the full version of this paper.

6 Expressive Power of TAX

In this section, we summarize our results on the expressive power of TAX. Details are provided in the full version of the paper.

**Theorem 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 $Q'(\text{Rep}(D)) = \text{Rep}(Q(D))$. 

A central motivation in designing TAX is to use it as a basis for efficient implementation of high level XML query languages. The following result identifies a subset of XQuery that is expressible in TAX.

**Definition 3 (Canonical XQuery Statement)** A canonical XQuery statement is a “FLWR” expression of XQuery [7] such that: (i) the variable declaration range in each FOR and LET clause is a path expression; (ii) there are no function calls or recursion in any expression; and (iii) all regular path expressions used involve only constants, wildcards and may further use ‘/’ and ‘//’.

**Theorem 2. (Canonical XQuery Translation)** Let $Q$ be a canonical XQuery statement such that no ancestor-descendant relationships that are not present in the input collection are introduced by $Q$. Then there is an expression $E$ in TAX that is equivalent to $Q$.

Similar translation theorems can be shown for Quilt and XML-QL.

7 Related Work

There is no shortage of algebras for data manipulation. Ever since Codd’s seminal paper [8] there have been efforts to extend relational algebra in one direction or another. We mention a few relevant ones here, and relegate a detailed discussion to the full version of the paper.

Gyssens et al. [13] present a grammar-based algebra for manipulating tree-structured data, and show that it is 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 (see, e.g., [14]). These ideas have been incorporated into an object-oriented database, and an algebra developed for these in the Aqua project [20]. 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.

Algebras and query languages have also been proposed over graphs (see, e.g., [10, 18, 1, 5]). These algebras focus on pattern matching in graphs. Since trees are a special case of graphs, our notions of pattern tree match may appear at first glance to be a special case of these works. However, there are differences in complexity of evaluation and in several details, such as the notion of order so important to XML trees. Moreover, in graphs there is no simple
8 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 few operators more than relational algebra. Furthermore, each of its operators uses the same basic structure for its parameters.

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.

Acknowledgements

The work of Jagadish and Thompson was supported by NSF under grant IIS-9986030, while that of Lakshmanan was supported by NSERC (Canada) and a seed grant from IIT - Bombay.
References