TIMBER: A Native XML Database

H. V. Jagadish[1], Shurug Al-Khalifa[1], Adriane Chapman[1], Laks V. S. Lakshmanan[2], Andrew Nierman[1], Stelios Paparizos[1], Jignesh M. Patel[1], Divesh Srivastava[3], Nuwee Wiwatwattana[1], Yuqing Wu[1], Cong Yu[1]

[1] University of Michigan, Ann Arbor, MI, USA*
e-mail: {jag, shurug, apchapma, andrewdn, spapariz, jignesh, nuwee, yuwu, congy}@umich.edu
[2] University of British Columbia, Vancouver, BC, Canada**
e-mail: laks@cs.ubc.ca
[3] AT&T Labs Research, Florham Park, NJ, USA
e-mail: divesh@research.att.com

The date of receipt and acceptance will be inserted by the editor

Abstract This paper describes the overall design and architecture of the Timber XML database system currently being implemented at the University of Michigan. The system is based upon a bulk algebra for manipulating trees, and natively stores XML. New access methods have been developed to evaluate queries in the XML context, and new cost estimation and query optimization techniques have also been developed. We present performance numbers to support some of our design decisions.

We believe that the key intellectual contribution of this system is a comprehensive set-at-a-time query processing ability in a native XML store, with all the standard components of relational query processing, including algebraic rewriting and a cost-based optimizer.

1 Introduction

With the growing popularity of XML, it is clear that large repositories of XML data will emerge. In this paper, we describe the architecture of Timber, a native XML data management system being developed at the University of Michigan [67].

One popular technique for managing XML data is to map the data to existing (relational) database systems. However, such a mapping often results in either an unnormalized relational representation or in a very large number of tables, due to the flexible nature of XML, with attributes and sub-elements frequently missing, and repetition of sub-elements being allowed.

Our approach in Timber is to start from scratch and develop an XML data management system from the ground up. Many components of a standard database system can be reused with no change. For instance there is no need to modify transaction management facilities. However, other components must be modified to accommodate the new data model and query language. The overall architecture of the Timber system is presented in Sec. 3.

Our challenge is to develop a native XML database, in which XML data is stored directly, retaining its natural tree structure. At the same time, we would like to obtain all the benefits of relational database management, such as declarative querying and set-at-a-time processing.

To be able to obtain efficient processing on large databases, we require set-at-a-time processing of data. In other words, we require a bulk algebra that can manipulate sets of trees: each operator on this algebra would take one or more sets of trees as input and produce a set of trees as output. We have devised such an algebra, called TAX, and we present this in Section 4. The biggest challenge in devising this algebra is the heterogeneity allowed by XML, and in XQuery [10], the W3C recommended declarative language for querying XML databases today.

Given an algebra, we need an efficient query evaluation mechanism. This is the subject of Section 5. After describing the overall structure of the query pipeline, we delve into a couple of access methods of significance.

* Supported in part by the United States National Science Foundation (NSF), under grants IIS-9986030, DMI-0075447, and IIS-0208852, and by an equipment grant from IBM.

** Supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and a research fellowship from the British Columbia Advanced Systems Institute (BCASI).

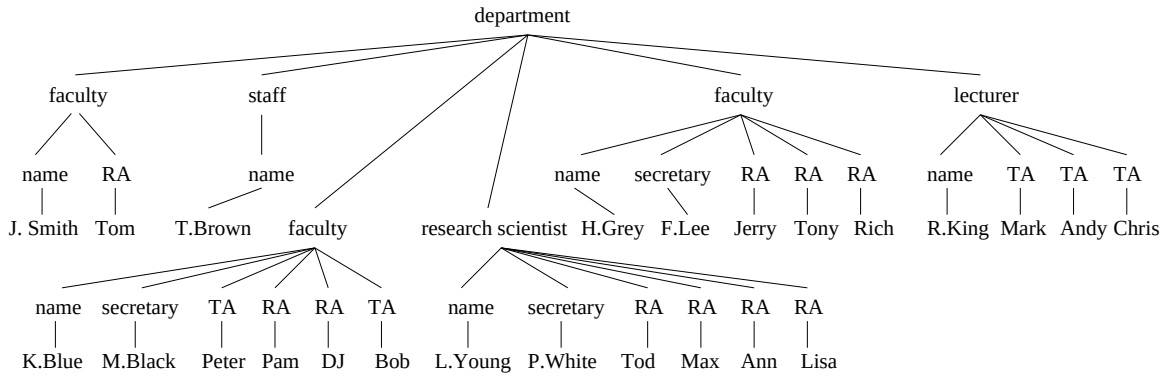

Fig. 1 Tree Representation of an Example XML document, *D*

A query optimizer is able to take a declarative query specification, and choose a suitable evaluation plan using the available access methods, making use of cost estimates for various operations and algebraic identities. We present the architecture of our optimizer in Section 6. We also present novel techniques for obtaining size (and cost) estimates.

After a brief discussion of issues regarding updates in Section 7, we finally wrap up with a discussion of the current status, and some indications of performance, in Section 8. We begin by setting the context for our work in Sec. 2

2 Motivation and Related Work

Example 1 *Figure 1 shows a very simple XML document. The personnel of a* department *can be* faculty, staff, lecturer *or* research scientist*. Each of them has a* name *as identification. They may or may not have a* secretary*. Each* faculty *may have both* TA*s and* RA*s. A* lecturer *can have one or more* TA*s, but no* RA*. A* research scientist *can have any number of* RA*s, but no* TA*.*

Some characteristics of XML data are obvious even from this simple example. XML has a tree structure: elements in the document can be structurally related and these structural relationships are meaningful. XML also has flexibility – the number of RAs and TAs associated with personnel is allowed to vary. While there are constraints on what is allowed, it is still quite possible for certain classes of sub-elements to be missing altogether. For instance, there may be a lecturer who has no teaching assistants at all.

Several mapping techniques have been proposed [21, 30, 48, 49] to express tree-based XML data to flat tables in a relational schema. Due to the possible absence of attributes and sub-elements, and the possible repetition of sub-elements, XML documents can have a very rich structure, as we just saw. It is hard to capture this structure in a rigid relational table without dividing the document into very small standard "units" that can be represented as tuples in a table. Therefore, a simple XML schema often produces a relational schema with many tables. Structural information in the tree-based schema is modeled by joins between tables in the relational schema. XML queries are converted into SQL queries over the relational tables, and even simple XML queries often get translated into expensive sequences of joins in the underlying relational database.

Example 2 *A typical translation [48] of the schema of figure 1 would map the* lecturer *elements to a table, and store* TA *elements as tuples in another table. To find the TAs assisting a specified lecturer will then require a join between the two tables. More complex queries will require multiple joins.*

Driven by the arguments above, one is persuaded to seek a direct implementation of XML data management, where XML data is not translated into rigid relations. There are several implementations of XML storage that are independent of relational databases [38, 45, 60, 62, 63]. Several of these are driven by the document (or programming language) community, rather than the database community. The implementations are procedural, directly evaluating queries as a series of nested FOR loops. They are also tuple-at-a-time, whereas it has been well established through the experience of the database community that set-at-a-time access is essential for good performance. As such, these implementations do very well for small data sets, but do not scale very well to large data sets. For instance, Xindice (see dbXML) recommends [4] that its system not be used for documents larger than 5MB!

Other solutions have also been proposed. For instance, XML databases have been implemented on top of an object-oriented database [17, 32, 44, 61] and a semi-structured database [34,35,42]. Such implementations suffer from a combination of the drawbacks listed above for the two extreme scenarios. Tamino is a leading commercial "native" XML database, yet descriptions of its architecture [46, 47] are fairly sketchy. Tamino uses an evolution of the ADABAS nested relational engine as its data store, with the bulk of the innovation in the product coming from new index structures, support for handling XML schematic information, and the web interface layer.

Recently, Natix [26, 27] has been developed as a storage manager suitable for XML data. The focus is on efficient management of tree-structured data at the level of page allocation and physical placement. Whereas our current development is on top of the more "standard" Shore storage manager, we intend to consider switching to Natix as the latter matures.

Our project is aimed centrally at building an efficient XML database engine. As such it differs from related efforts at data integration [6, 52] and querying XML over the web [40]. However, each of these important research efforts requires at least some management and querying of XML data as part of their research effort. As such, each is exploring issues that closely relate to ours in many cases. For instance, we will mention techniques used in the Niagara [40] system at several places below.

Finally, we mention the Toronto XML project [5], aimed at managing XML data using an approach complementary to ours. Whereas we are developing new techniques for managing and querying tree-structured XML data, the Toronto project maps XML into flat files, RDBMS or OODBMS, whichever is most appropriate for a given class. The core of their effort is in managing the metadata for this mapping and in developing clever new index structures for this heterogeneous representation.

3 System Architecture

The overall architecture of Timber is shown in figure 2. We build our system on top of Shore [9], a popular back-end store that is responsible for disk memory management, buffering and concurrency control. XML data, index and metadata are also stored in Shore through Data Manager, Index Manager and Metadata Manager, respectively.

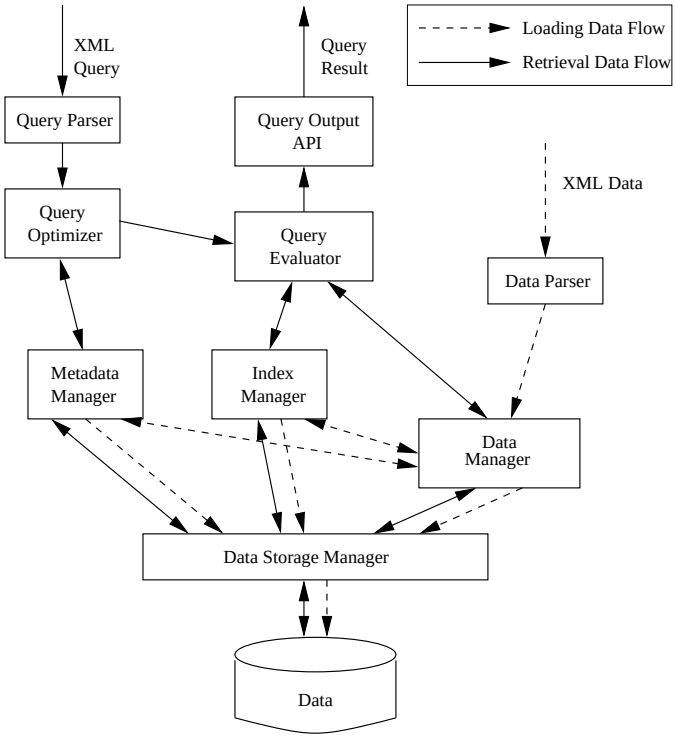

Fig. 2 TIMBER Architecture Overview

3.1 Data Storage

The Data Parser takes an XML document as input, and produces a parse tree as output. The Data Manager takes each node of this parse tree as it is produced, transforms it incrementally into an internal representation and stores it into Shore as an atomic unit of storage.[1] A set of navigation interface and scan interface is provided for the Query Evaluator to retrieve data, one node at a time. These interfaces can also be used by Index Manager and Metadata Manager, to generate the data they need.

For storage efficiency reasons, a node in the Timber Data Manager is not exactly the same as a DOM [57] node. There is a node corresponding to each element, with child nodes for sub-elements. However, all attributes of an element node are clubbed together into a single node, which is then stored as a child node of that element node. Also, the content of an element node, if any, is pulled out into

[1] We found that Shore had considerable overheads in dealing with small objects. We are engineering our system to package our data in page-size containers, and handing Shore an entire container as an object. At present, this engineering optimization has been implemented in our Query Evaluator for intermediate results that may have to be read and written multiple times in quick succession. This optimization is less critical for the actual data itself, and has not yet been implemented in the Data Manager.

a separate child node. If the node is of mixed type, with multiple content parts interspersed with sub-elements, each content part is pulled out into a separate child node. Finally, due to our focus on data management issues, all processing instructions, comments, and such are simply ignored. In a future version of our system, we could create yet another child node of the element node with all such data.

An inclusion relationship between an element and its sub-elements is the tightest possible bond between two entities in a database. Entire sub-trees are frequently requested. In fact, in a document representation of the database, a sub-tree corresponds to a contiguous fragment of the document. As such, the determination of parent–child and ancestor–descendent containment relationships is a very frequent operation in XML query processing. It has been observed [2, 13, 39] that it is possible to associate a numeric start and end label with each data node in the database, defining a corresponding interval between these labels such that every descendant node has an interval that is strictly included in its ancestors' interval. If each node is also labeled with its Level, or nested depth of the node in the document, then parent–child relationships can also be found. The relevant formulae are:

- Ancestor–descendant relationship: a node (S_1, E_1, L_1) is the ancestor of node (S_2, E_2, L_2) iff $S_1 < S_2 \wedge E_1 > E_2$.
- Parent–child relationship: a node (S_1, E_1, L_1) is the parent of node (S_2, E_2, L_2) iff $S_1 < S_2 \wedge E_1 > E_2 \wedge L_1 = L_2 - 1$.

(S_1 and S_2 are start labels, E_1 and E_2 are end labels, and L_1 and L_2 are level labels in these formulae.)

We will discuss, in Sec. 5, how we use these formulae in Timber. For the present, we focus on how these start, end and level labels are managed. Conceptually, these labels are additional attributes created automatically by the system and associated with each node. Where document boundaries are important, one could create separate labels for each document, so that an additional doc label would be required to match in addition to the interval subsumption described above. It is easy to map between such a multi-document model, and a model in which the ranges of label values for each document are assigned to be non-overlapping, doing away with the need for a separate doc attribute.

Updates are an issue in any such labeling scheme, see [14]. It is conceivable that a complete re-labeling could be required for each update, leading to very poor update performance. We address this issue by leaving gaps between successive label values. With this mechanism, relabeling is required only if a large number of insertions take place within the same small label value range. If updates are well distributed, no relabeling may be required for a long time. See [13]. We use double values for these labels in the current version of Timber, as an "automatic" means of leaving gaps, at least to within machine precision. Note that as new data is appended (as opposed to being inserted in the middle), new larger label values can simply be manufactured for the appended nodes with no effect on the existing nodes.

In relational databases, a record identifier (typically called an "rid" or a "tid") is used to identify each record. This is not quite an identifier in the sense of an object-oriented database – there is no concept of object identity. It frequently is a function of physical placement of the record (like a physical pointer), but it does not have to be: it is truly a logical identifier. It is also not visible to the user at the query level. Nonetheless, it plays a central role in relational query processing. For an XML database, we seek a corresponding *node identifier*. XML permits an optional ID attribute, but this is not quite it, since this is user-visible, and is optional, and further is not even applicable for nodes that do not correspond to XML elements (such as attributes and comments). The normal solution would be to invent such an identifier for our system. However, we find that the tuple of start, end, and level labels serves this purpose admirably. As such, we shall use this triple of labels as node identifier. Note that while start alone suffices to serve as a node identifier, using the triple as a node identifier enables efficient index-based query processing, as we'll see later.

The physical storage order of XML elements can significantly impact the cost of data access. Since we expect sub-elements to be requested frequently with an element, ideally we would like to cluster these together. It is generally believed that storing XML data in document order (or pre-order tree traversal order) is the most desirable. This is what we do. An equivalent way of expressing this is that we would like to store our nodes sorted by the value of their start labels. Again, updates are an issue. See Sec. 7.

3.2 Index Storage

There is a rich history of work on index structures suited to specific purposes. In particular, we draw inspiration from the work done in the context of object-oriented systems, such as [7, 29]. More recently, novel path indices have been proposed for XML and semi-structured data [15, 28, 37]. Schema summarization structures have also been proposed [23, 24]. We are intensively studying this problem, but at the current time have only single-node indices implemented in Timber.

We construct value indices on attribute values, whether these are numeric or character string. We also construct indices on element content, when this content is recognized as a number. We also construct term-based inverted indices on element content when this is a large piece of text. In addition, we construct an index on tag name: that is, given a tag name, we can return all the elements with the specified tag. All our indices are stored using the B-Tree index facility provided by Shore.

Index structures typically return a list of *Rid*s in relational systems. Correspondingly, they return lists of start, end and level labels in an XML database.

3.3 Metadata Storage

Timber has a metadata store that is, for the most part, not remarkable. There is the usual information regarding attribute types, data set sizes and indices constructed. The histograms maintained for cost estimation purposes are novel, and are described in Sec. 6.

Schema plays a crucial role in traditional databases, and table structure is a crucial part of the metadata maintained. However, in the design of XML, much care has been taken to make sure that a great deal can be accomplished even in the absence of schema (or DTD).[2] In the same spirit, we have designed the core of Timber not to have any dependence on schema whatsoever. The bulk of the description in this paper is with regard to the Timber core, and hence has little mention of schema.

Knowledge of schema can play an important role in data layout, in choice of index structures, and in query optimization. Our goal is to use this information, when available, to advantage; while continuing to retain reasonable performance even when schema information is not available. For instance, even data statistics are collected in our position histograms (described in Sec 6 below), without specific reference to the schema.

3.4 Query Processing

XML queries in XQuery [10][3] are parsed into an algebraic operator tree by the Query Parser. (The tree algebra used for this purpose is described in Sec. 4). The Query Optimizer reorganizes this tree, based on a set of rules and metadata information, and performs the required mapping from logical to physical operators. The resulting query plan tree is evaluated by the Query Evaluator, pipelined one operator at a time, by means of a set of calls to the Data Manager and Index Manager, which in turn call Shore storage.

[2] In fact, there is not yet complete agreement with regard to the best means of expressing XML schema information [64, 65].

[3] We have designed Timber to be as language independent as possible. We have written parsers for other languages, including Quilt [11], XML-QL [16], and XQL [43], but no longer maintain these.

4 Tree Algebra

An XML document is a tree, with each edge in the tree representing element nesting (or containment). See figure 1, for example. Structural relationships in this tree are central to most XML querying. As such, an appropriate algebra for XML should manipulate sets of trees. That is, each operator in the algebra should take as input one or more sets of trees and produce as output a set of trees.

Order is important to XML documents. As such, the trees manipulated by the algebra should be ordered. (This is true, even if queries frequently do not care about the order. See labeled paragraph on ordering later in this section.) Moreover, each node in a tree represents an XML element, and is thus labeled with the element tag and any attributes of the element. In short, we require an algebra to manipulate sets of ordered labeled trees.

XML also permits references, which are represented as non-tree edges, and may be used in some queries. These are important to handle, and our algebra is be able to express these. However, there is a qualitative difference between these reference edges, which are handled as “joins”, and containment edges, which are handled as part of a “selection”.

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

Heterogeneity: Each tuple in a relation has identical structure – given a set of tuples from some relation in relational algebra, we can reference components of each tuple unambiguously by attribute name or position. Trees have a more complex structure than tuples. More importantly, sub-elements can often be missing or repeated in XML. As such, it is not possible to reference components of a tree by position or even name. For example, in a bibliographic XML tree, consider a particular book sub-tree, with nested (multiple) author sub-elements. We should be able to impose a predicate of our choice on the first author, on every

author, on some (at least one) author, and so on. Each of these possibilities could be required in some application, and these choices are not equivalent.

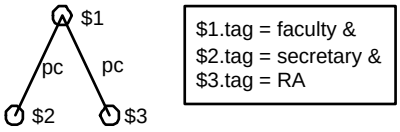

Fig. 3 Pattern Tree, P, for a Simple Query

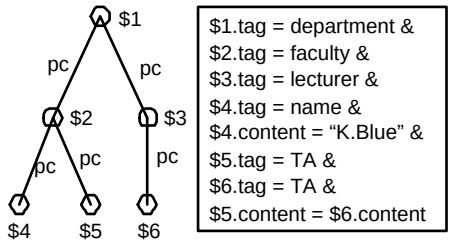

Fig. 4 Pattern Tree, P', for a Less Simple Query

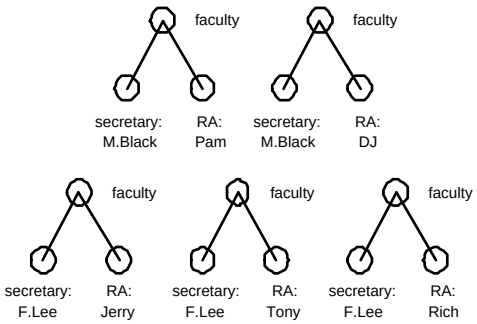

Fig. 5 Witness Trees for the Pattern P of Figure 3

We solve this problem through the use of *pattern trees* to specify homogeneous tuples of node bindings. For example, a query that looks for faculty members who have both a secretary and an RA can be expressed by a pattern tree shown in figure 3. Matching the pattern tree to the example database, the result is the sub-trees, which are rooted at element “faculty” and have two child elements, “secretary” and “RA”. From the example XML document in figure 1, we can see that the sub-trees for faculty “K.Blue” and “H.Grey” will be selected, as shown in figure 5. Such a returned structure, we call a *witness tree*, since it bears witness to the success of the pattern match on the input tree of interest. One witness tree is produced for each combination of node bindings that matches the pattern. The set of witness trees produced through the matching of a pattern tree are all homogeneous: we can name nodes in the pattern trees, and use these names to refer to the bound nodes in the input data set for each witness tree. A vital property of this technique is that the pattern tree specifies exactly the portion of structure that is of interest in a particular context – all variations of structure irrelevant to the query at hand are rendered immaterial. In short, one can operate on heterogeneous sets of data as if they were completely homogeneous, as long as the places where the elements of the set differ are immaterial to the operation.

Conditions other than tag names may be associated with pattern trees. Figure 4 shows a more complex pattern tree that places a number of additional conditions on the nodes participating in the pattern. Node $2 can only be matched by a faculty whose name is “K.Blue”. Furthermore, this faculty is required to have a TA (at node $5) who is also a TA (at node $6) to some lecturer (node $3) in the same department (node $1).

XPath is very popular, and is frequently used in place of XQuery for XML query processing. Also, the crucial variable-binding FOR clause (and also the LET clause) of XQuery uses a notation almost identical to XPath. So it is worth spending a moment to see how the notion of pattern tree relates to an XPath expression. The key difference is that one XPath expression binds exactly one variable, whereas a single pattern tree can bind as many variables as there are nodes in the pattern tree. As such, when an XQuery expression is translated into the tree algebra, the entire sequence of multiple FOR clauses can frequently be folded into a single pattern tree expression.

All operators in TAX take collections of data trees as input, and produce a collection of data trees as output. TAX is thus a “proper” algebra, with compositionality and closure. The notion of pattern tree plays a pivotal role in many of the operators. Below we give a sample of TAX operators by describing briefly how selection, projection and grouping work. Further details and additional operators can be found in [25].

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 rel-

evant witness tree(s) rather than just a single bit with each data tree in the input to the selection operator.

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

Consider once more the example database of figure 1 and the pattern tree shown in figure 3. A selection using this pattern tree, P, and an empty adornment list, on the example database, D, would be written $\sigma_{P,\{\}}(D)$. One expects that the outcome would be the faculty members of interest (K.Blue and H.Grey), and possibly the sub-tree rooted at each. But it is not enough to return the input database tree in the output as satisfying the selection "predicate". In relational algebra, selection simply filters elements of a set – the output of a selection operator is a subset of its input. In a tree algebra, selection does more than filter since it identifies the relevant matching portion of the input document (set element). Where multiple matches occur, each match is shown separately in the output, as in figure 5. Information retrieval systems sometimes highlight search terms in the retrieved documents: our proposal takes this idea one step further for selection queries in a tree algebra.

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

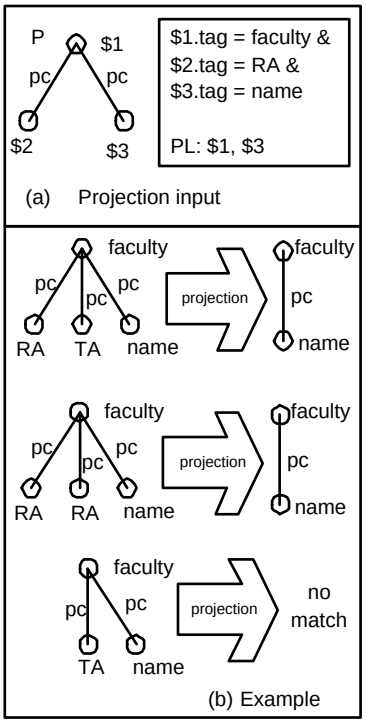

Fig. 6 A sample projection operator $\pi_{P,PL}(C)$. (a) shows the input pattern tree P and projection list PL. (b) shows an example application on two different input trees. To minimize clutter, labels have been dropped from ad edges in the pattern tree. pc edges are labeled.

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.

A simple projection example is shown in figure 6a. Part (b) for this figure shows how this projection would apply in three cases. The first faculty member has an RA, a TA, and a name; the pattern tree match is straightforward; and the projection result is what one would expect. The second faculty member has two RAs, and hence has two separate witness trees that would match the sepcified pattern tree. Both these witness trees are identical with respect to the projected elements ("faculty" and "name"). As such, only one result is produced. This is duplicate elimination by "identifier", and is used by all TAX operators to remove gratu-

itous duplicates, as in this example. Note that this is different from duplicate elimination by value, where we notice identical values for the names and other attributes of two different faculty members, and hence remove one of them. The latter operation is ptentially expensive, and carried out only upon explicit request. The former operation can actually be used to reduce the cost of operator evaluation, as shown in [3]. The third faculty member in the figure has no RAs, and hence produces no results on account of no pattern tree match. This is so, in spite of the fact that the second faculty member does have all the attributes retained in the projection.

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.

Ordering: As noted above, trees in XML are ordered. However, queries often do not care about this order. As such, we need to allow for pattern trees that match while preserving order, and pattern trees that do not necessarily preserve order when matching. Rather than introduce one additional choice variable, we specify pattern trees to be unordered except where ordering constraints are explicitly specified. Even for a completely ordered tree, we can show that the additional length of the pattern tree specification does not asymptotically increase the size of pattern tree description. The reason is that order is a transitive notion, so only the transitive reduction of the ordering needs to be specified. In the case of total ordering of n nodes, this requires $n-1$ order relations between immediate successors. A benefit of our approach is that ordering constraints can be specified selectively where they matter in a pattern tree.

Sets, by definition, are unordered. In SQL, we often require the answer set to be sorted by some criterion. This sorting is not part of the relational algebra – instead it is performed at the end, as part of the output. In our algebra, trees are ordered while sets are unordered, so we have a greater richness, and it actually becomes possible to incorporate sorting (and ordering operations in general) as part of the algebra. Specifically, an unordered set of trees can be combined into a single tree by ordering the set of trees and then making each an immediate sub-tree of a new root node.

XQuery permits elements to be ordered according to "document order". In fact, this is the default order expected if none other is specified. We use the start label of a node for this purpose.

Grouping: In relational databases, tuples in a relation are often grouped together by partitioning the relation on selected attributes – each tuple in a group has the same values for the specified grouping attributes. Given the more complex structure of trees, there may be a good reason to group based on some arbitrary function of each tree rather than a simple equality on selected attributes. For instance, we may wish to group faculty in the example of figure 1 based on the number of RAs associated with the faculty member. These numbers are never explicitly stored in the database anywhere, and are themselves obtained as the result of a "structural aggregation". For another example, books in a bibliographic database may be grouped based on the state of residence of the first author.

A source of potential difficulty is that grouping may not induce a partitioning due to repeated sub-elements. If a book has multiple authors, then grouping books by author will result in this book being repeated as a member of multiple groups.

A deeper point to make is that grouping and aggregation are not part of relational algebra, though they are important physical operators in relational database systems. The reason is that these operators cause a "type violation": a grouping operator maps a set of tuples to a set of sets of tuples, and an aggregation operator does the inverse. The flexibility of XML permits grouping and aggregation to be included within the formal tree algebra, at the logical level.

We formalize this as follows. The groupby operator $\gamma_{P,gb,ol}(\mathcal{C})$ takes a collection $\mathcal{C}$ as input and the following parameters. A pattern tree $\mathcal{P}$; this is the pattern used for grouping. A *grouping basis* gb that lists elements by label in $\mathcal{P}$ (and/or attributes of elements), whose values are used to partition the set $\mathcal{W}$ of witness trees of $\mathcal{P}$ against the collection $\mathcal{C}$. Element labels may possibly be followed by a '*'. An *ordering list* ol, each component of which comprises an *order direction* and an element or element attribute (specified by label in $\mathcal{P}$), with values drawn from an ordered domain. The order direction is either ASCENDING or DESCENDING. This ordering list is used to order members of a group for output, based on the values of the component elements and attributes, considered in the order specified.

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

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

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

then the corresponding child is a match of this node. If the item is `$i*`, then in addition to the said match, the subtree of the input tree rooted at the matching node is also included in the output. (b) Its right child r has tag `tax_group_subroot`. Its children are the roots of input trees in $\mathcal{C}$ that correspond to witness trees in $\mathcal{W}_i$, ordered according to the ordering list. Input trees that produce more than one witness tree will appear more than once.

Following the principles outlined above, we have developed TAX, a tree algebra for XML. The operators are selection, projection, product, set union, set difference, renaming, reordering, and grouping. Details can be found in [25]. It has been shown that the core of XQuery can be expressed in terms of TAX operators. The first step in the Timber system is to parse a given XQuery expression to obtain an equivalent TAX expression, which can subsequently be optimized using algebraic identities.

A frequent case is when we rephrase XQuery expressions written as nested FLWR clauses into simple ("single-block") tree algebra expressions involving grouping. The following example demonstrates how this works. Details of the described algorithm can be found in [41]. Let's consider a sample nested FLWR statement, as seen in Query 1 in Figure 7.

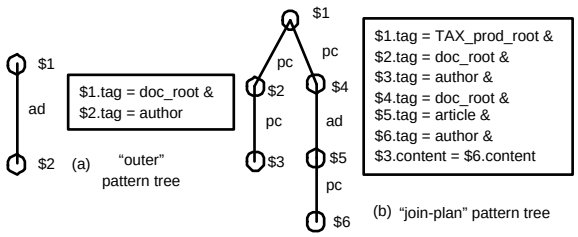

Fig. 8 The generated selection pattern trees of a naïve parsing of query 1 in figure 7.

A naïve translation of this would produce an inefficient nested FOR loop. The outer combination of FOR/WHERE clauses will generate a pattern tree ("outer" pattern tree). A selection will be applied on the database[4] using this pattern tree; the selection list consists of the bound variables in XQuery. For Query 1 the pattern tree is shown in Figure 8.a. The selection list is $2.

The inner combination of FOR/WHERE clauses will generate a pattern tree that describes a left outer join between all the authors of the database, as selected already and bound to variable $a, and the authors of articles. This pattern tree is shown in Figure 8.b. A left outer join is generated using this pattern tree and applied on the outcome of the "outer" selection and the database. It uses a selection list $5. Following this join operation there will be a projection with projection list $5* and then a duplicate elimination based on articles.

To produce the final result the necessary stitching will take place using a full outer join and then a renaming to generate the tag name for the answer.

With the use of grouping, we can produce a simpler and more efficient execution. We present next the outline of an algorithm to detect the naïve execution, and rewrite it more efficiently with the grouping operator.

1. Construct an initial pattern tree from the "inner" FLWR statement and consisting of the bound variables and their paths from the document root, including any conditions that apply to these variables without reference to variables bound in the outer loop. For Query 1 this pattern tree is seen in figure 9a. We apply a selection using this pattern tree with selection list the elements corresponding to the bound variables and a projection with a projection list similar to the selection list. For Query 1 those lists will be $2 and $2* respectively.
2. Construct the input for the GROUPBY operator.
   - The *input pattern tree* is generated from the join plan pattern tree of naïve parsing. It consists of the bound variable of the "inner" statement and the node where the join was specified. For Query 1 this is shown in figure 9b.
   - The *grouping basis* will be the join value of the nested query. For Query 1 this will correspond to the author element or $2.content in the group by pattern tree of figure 9b.
3. Apply the GROUPBY operator on the collection of trees generated from step 1. This will create intermediate trees containing each grouping basis element and the corresponding pattern tree matches for it. For Query 1 the tree structure will be as in figure 9c.

[4] The database is a single tree document

4. A projection is necessary to extract from the intermediate grouping tree the nodes necessary for the outcome. The projection pattern tree is generated from each argument of the RETURN clauses. For query 1 this is shown in figure 9d.
5. After the final projection is applied the outcome consists of trees with an dummy root and the authors associated with the appropriate titles. A rename operator is necessary to change the dummy root to the tag specified in the return clause.

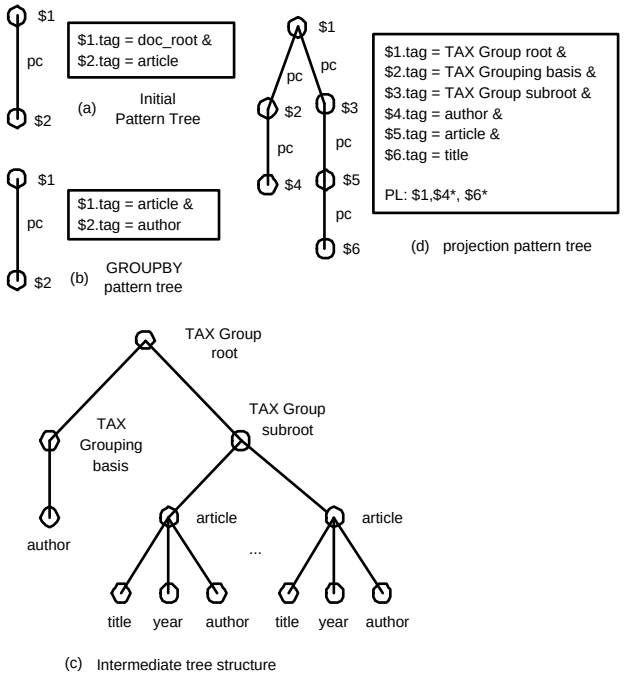

Fig. 9 GROUPBY operator for Query 1. The generated input and the intermediate tree structure

5 Query Evaluation

5.1 Physical Algebra

In the relational world, there is an important distinction between the logical algebra and the physical algebra. The former includes cartesian product, for example, as a core operator, and does not permit sorting. The latter includes natural join and sorting as core operators. Moreover, the latter manipulates ordered sets (and exploits ordering), whereas the former only deals with unordered sets. It stands to reason that there are similar needs in XML databases as well.

In addition we have the issue of determining how to reconcile pattern tree matching at the logical level with nodes being the atomic unit of data storage. In a relational system, the unit of logical operation is the same as the unit of physical operation. In XML, we are logically manipulating trees, but physically manipulating "node-structures". As such, the physical algebra for Timber has greater separation from the logical algebra than in relational systems. In particular, data is accessed at the granularity of nodes, and indexing is performed at the granularity of nodes. Furthermore, the root nodes of a tree can frequently be used in place of the tree itself for query processing.

The bulk of the physical algebra is relatively mundane, with all the operators one would normally expect, such as joins, selections, sorting, and so forth. In the interest of space, we skip these details here and refer the interested reader to [67]. Instead, we describe below two features that are particularly noteworthy. One is the reuse of pattern trees. The other is the explicit physical operator for data materialization.

Pattern Tree Reuse: Given a heterogeneous set of trees, we use pattern tree matches to identify nodes of interest: the nodes to which conditions apply, the nodes that should be manipulated, etc. Thus, as described in Section 4, most (logical) tree algebra operators require a pattern tree as a parameter. In an algebraic expression, it is frequently the case that multiple operators use exactly the same pattern tree. It is computationally profligate to re-evaluate the pattern tree each time for each operator. Instead, we permit a pattern tree evaluation to be pulled out as a distinct physical operator (sequence), the results of which persist, and can be shared with many of the subsequent operators. For example consider pattern tree 1 in figure 10. We can apply a selection using this pattern tree and selection list $2, then a projection with the same tree and projection list $2, $4. The selection operator returns a set of faculty who have both RA and name children, along with the entire sub-tree rooted at each. The projection operator retains only the faculty and name nodes from each sub-tree.

Persistence of pattern tree matches is accomplished through the use of a pattern tree identifier (PID) and a witness node identifier (WID) within the tree. Every database node that could serve as a match for a particular witness node position in a particular pattern tree has the corresponding "PID-WID" recorded as part of the intermediate result. Subsequent operations that use the pattern tree can then refer to the set of all nodes carrying the corresponding PIDWIDs. For instance, a node selection predicate physical operator can be applied to node $3 of pattern tree numbered 2, by applying the predicate to all nodes in the node-structure input to this operator with a PIDWID of $(2, 3)$.

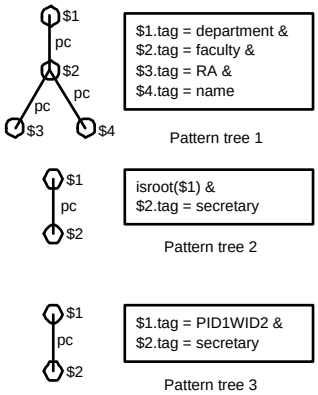

Fig. 10 Sample pattern trees. Pattern tree 3 is an extension of pattern tree 1.

One can think of pattern tree reuse as akin to common sub-expression elimination. A complication to consider in the case of pattern tree reuse is that operators actually manipulate tree structure. A structural pattern matched before a particular algebraic operator may no longer match after the operator, and vice versa. Even worse, it is possible for the pattern to match, but now bind to different nodes. For example, consider pattern tree 2 in figure 10. If a projection is applied on the database using this tree and projection list $2, the empty set will be returned since no secretary is a direct child of the root node in the database of figure 1. But what if a selection is applied on the database first, returning all faculty and their child nodes. Then a projection using pattern tree 2 will return every secretary in the database, since each is directly below some faculty, returned as the root of a tree in the output of the selection.

Consider a join predicate to be applied to a pair of nodes, each of which has been identified by means of a distinct pattern tree. This too is easily specified, using the PIDWIDs of the corresponding nodes: the fact that separate pattern trees were used to identify each node makes no difference. In fact, all the logical algebra operators, except for grouping, preserve (relevant portions of the) tree structure, and hence permit the use of persistent PIDWIDs, provided that all node predicates are quantifier-free and only reference node tags, identifiers, and attribute values. Notably, this includes the cartesian product operator.

Sometimes, subsequent operators in a logical algebra expression may not use the exact same pattern tree, but rather may use a variation of it. Our PIDWID scheme permits *pattern tree extension*. We can reference a previously computed pattern tree match, and apply additional conditions to the node-structures known to satisfy the original match. These additional conditions are in the form of an additional pattern tree that references previously matched nodes in common with the original using their PIDWIDs. For example we apply a selection on the database using pattern tree 1 of figure 10 and selection list $2. Then we want to apply a projection to find out the secretary for each faculty member. There is no need to create a new pattern tree with complicated structure for this purpose. We reuse pattern tree 1 and we extend it to generate pattern tree 3 using a PIDWID reference. Then a projection can be applied using pattern tree 3 and $2 as the projection list. Note that the secretary element could not have been included in pattern tree 1 to begin with: the applied selection would have produced different output. (The output would have been restricted to faculty who have RA, secretary and name, rather than including faculty with RA and name but no secretary).

Node Materialization: In relational databases, conjunctions of selection conditions are often evaluated through intersection of rid sets, obtained from indices, without accessing the actual data. However, for the most part, query evaluation does process the actual data in the evaluation pipeline. In the case of XML trees, it is possible to encode the tree structure (see discussion of start and end attributes in the next section) so that quite complex operations can be performed without accessing the actual data itself. On the flip side, the actual data itself is a well-circumscribed tuple in the case of a relational database. But for an XML element, we may be interested in the attributes of this element itself, in its child sub-elements, or in its entire descendant sub-tree: which depends on the context. As such, at the physical level, it is important to distinguish between identification of a tree node (XML element), by means of a node identifier, and access to data associated with this node. Consequently, we have an explicit materialization operator in the physical algebra. This operator takes a (set of) node identifier(s) as input and returns a (set of) XML tree(s) that correspond.

In an XML database, as in any other database, we use indices to find portions of the database relevant to a query whenever possible. An index lookup returns a list of node identifiers. In a relational database the corresponding tuple identifiers (or “rid”s) would be dereferenced (almost) immediately. However, considerable additional processing may be possible, in the case of XML, based purely on the node identifiers. Consequently, during query processing, we keep only the ids of nodes around as far as possible. We call such intermediate results *unmaterialized*.

Of course, there will be operations for which access to the data is necessary. But now there is the question of what “the data” corresponding to a node is. We may need only the value of one attribute for some predicate evaluation or grouping. Or we may need data from a child sub-element.

And so on. A reasonable technique is to materialize exactly the minimum amount required, and work with intermediate results that are *partially materialized*. By so doing, we minimize the size of intermediate results being manipulated.

An option at the other extreme is to *fully materialize* each node identifier immediately – obtaining all the data associated with it (and its sub-tree, if need be). As stated above, this option is usually very expensive.

As a small example consider pattern tree 1 of figure 10. A simple query consists of a selection using this pattern tree and then a projection using the same pattern tree and $4 as projection list. "The name of each faculty member that has an RA". The only node that needs to be materialized is $4 (name) at the end of the query. Cases like these are very common and fully materializing everything is unnecessary.

5.2 Structural Joins in Pattern Tree Matching

Most logical algebra operators take a tree pattern as parameter. Every query plan that results has satisfaction of (at least one) tree pattern match as an early evaluation step. (There are two reasons for this. The syntactic reason is that there are no bound nodes to be manipulated until pattern trees have been matched. The performance reason is that the pattern tree match is akin to (a complex) selection, and is an important means to reducing the amount of data to be processed in the remainder of the query.) A construct that appears very often in a pattern tree is the *structural join* construct, which is used to specify a parent–child relationship or an ancestor–descendant relationship. Consequently, efficient implementation of the structural join is critical in determining the overall performance of an XML query processing system. We describe next, in some detail, our thoughts with respect to the implementation of structural joins for pattern tree matching.

A pattern tree, such as the one is figure 3 explicitly specifies predicates at nodes that must be satisfied by (candidate) matching nodes and also specifies structural relationships between nodes that match. Each edge in the pattern tree specifies one such structural relationship, which can either be "parent-child" (immediate containment) or "ancestor-descendant" (containment).

The simplest way to find matches for a pattern tree is to scan the entire database. Multiple matches of the pattern tree can share node bindings in common. Again, consider the example query in figure 3. Even though only two faculty members have both secretary and RA, the result contains five witness trees, for each pair of secretary and RA of the same faculty member. The five witness trees that will be returned share two different faculty-secretary pairs. As such, a naive scan algorithm will not be able to find all these matches in a single pass. An appropriate adaptations of effective pattern-matching techniques for strings (e.g Boyer-Moore [8], or KMP [31]) is required.

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

Experimentally, our own work [2], as well as that of others [59], has shown that under most circumstances it is preferable to use all the indices available and independently locate candidates for as many nodes in the pattern tree as possible. Structural containment relationships between these candidate nodes is then determined in a subsequent phase, one pattern tree edge at a time. For each such edge, we have a containment "join condition" between nodes in the two candidate sets. We seek pairs of nodes, one from each set, that jointly satisfy the containment predicate.

Example 3 *Consider a query, against the database D introduced in figure 1, seeking faculty who have a secretary reporting to them. The pattern to be matched has two nodes: a parent node that matches data nodes with tag* faculty*, and a child node that matches data nodes with tag* secretary.

A navigational access plan would start with a match at one of the two nodes in the pattern, and then navigate from it to find a match for the other node. For instance, there are three faculty nodes and three secretary nodes in the database. We could start from each of the three faculty nodes and explore all children to see if any of them is a secretary. When any such is found, the faculty-secretary pair can be returned as a witness tree. While the navigational effort involved is not huge in this small database for this trivial pattern, it is not hard to imagine that it could be very expensive given complex patterns, including indirect containment, to be matched on large databases.

A structural join access plan for the same pattern match task would first create lists of matches for each individual node in the pattern: namely the list of three faculty nodes and the list of three secretary nodes. Then it would perform a structural join to determine which faulty-secretary node pairs have a parent-child relationship.

Structural Join Algorithms Join is an expensive operation in a relational database. It tends to be the same in an XML

```
Algorithm Stack-Tree-Anc (AList, DList)
/* AList is the list of potential ancestors, in sorted order of StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

a = AList->firstNode; d = DList->firstNode; OutputList = NULL;
while (the input lists are not empty or the stack is not empty) {
   if ((a.StartPos > stack->top.EndPos) && (d.StartPos > stack->top.EndPos)) {
      /* time to pop the top element in the stack */
      tuple = stack->pop();
      if (stack->size == 0) { /* we just popped the bottom element */
         append tuple.inherit-list to OutputList }
      else {
         append tuple.inherit-list to tuple.self-list
         append the resulting tuple.self-list to stack->top.inherit-list
      }
   }
   else if (a.StartPos < d.StartPos) {
      stack->push(a)
      a = a->nextNode }
   else {
      for (a1 = stack->bottom; a1 != NULL; a1 = a1->up) {
         if (a1 == stack->bottom) append (a1,d) to OutputList
         else append (a1,d) to the self-list of a1
      }
      d = d->nextNode
   }
}
```

Fig. 11 Algorithm Stack-Tree-Anc with output in sorted ancestor order

database. Structural join computation is at the heart of tree pattern matching, which in turn is at the heart of XML query processing. Therefore, finding an efficient algorithm for evaluating a structural join is crucial.

Using the formulae in Sec. 3, each structural join is represented as an ordinary relational join with a complex inequality join condition. Variations of the traditional sort-merge algorithm can be used to evaluate this join effectively. Such variations have been suggested in [2,59]. However, one can exploit the tree structure of XML to do better. We have developed, and use in Timber, a whole *Stack-Tree* family of structural join algorithms. In figure 11 we describe one such algorithm that turns out to be used most frequently.

The intuition behind the algorithm is as follows. In a depth-first traversal of the database tree, every ancestor-descendant pair appears on a stack with the ancestor below the descendant. We exploit this observation to perform a limited depth-first traversal, skipping over nodes that are not in either input candidate list (AList or DList). (We use AList to denote the list of candidate ancestor (or parent) nodes. We use DList to denote the list of candidate descendant (or child) nodes.) We require an in-memory stack of size as large as the maximum depth of the XML document. The basic idea is to take the two input operand lists, AList and DList, both ordered by the Start position and merge them using a stack.

A node n with the smallest Start position is pulled from one of the input lists. If this Start is greater than the End of any node x already on stack, then we have finished traversing the portion of the tree involving x, and node x can be popped. If node n is from the AList it is pushed onto stack. If node n is from the DList, then it merges with each node in the stack to create a result pair. If the output were produced immediately, then the output would be sorted by the Start position of the descendant node in the join pair.

The sort order of operator output can be very important for pipelined query evaluation. It is typically most useful to have the results sorted by the Start position of the ancestor node in the join pair. In this case, a join result cannot be output until all the join results with ancestor nodes of lower Start value are output. This is done by keeping a list of join results with each of the ancestor nodes in the stack, appending the list to the next node in stack when one node

is popped, and outputting result only when the bottom of the stack is popped. Through careful list manipulation, we can perform this result-saving with limited memory buffer space and at most one additional I/O (write and then read back) for any result page.

Small variations of the algorithms described above can be used if the desired structural join is a parent-child (immediate containment) join rather than an ancestor - descendant (containment) join. Similarly, one can define semi-join, outer-join, and other variants. (Semi-joins, and left outer joins, in particular, seem to occur frequently in XML queries). Experiments show that these algorithms far outperform the navigation-based join algorithms, as well as the RDB implementation, in all cases.

The space and time complexity of the *Stack-Tree-Anc* algorithm is $O(|AList| + |DList| + |OutputList|)$. The I/O complexity is $O(\frac{|AList|}{B} + \frac{|DList|}{B} + \frac{|OutputList|}{B})$, where B is the blocking factor. (These asymptotic results apply to most other algorithms in the Stack-Tree family as well).

5.3 GroupBy

We discussed above that grouping does not necessarily partition the set. *E.g.*, the same book may have to appear in multiple groups, once for each author. RDBMS implementations of grouping typically rely on sorting (or possibly hashing). We cannot use these implementations directly. One possibility is for us to replicate elements an appropriate number of times, and to tag each replica with the correct grouping variables to use. For example, a two-author book would be replicated to produce two versions of the book node, with one author tagged in each replica as the one to use for grouping purposes. Thereafter standard sorting (or hashing) based techniques may be used.

The simple procedure suggested above requires cumbersome tagging, and involves needless early replication. Our implementation uses a slight variation that minimizes these disadvantages. The central idea is to recall that the grouping list (the list of variables on the basis of which to group) consists of nodes identified by means of a pattern tree match. The normal pattern tree match procedure will produce all possible tuples of bindings for these grouping variables. The sorting (or hashing) can then be performed using the resulting tuples of bindings.

If the grouping were to be followed by aggregation, as is frequently the case, this replication can be avoided altogether. For instance, suppose we are interested in the count of books written by each author. We can perform the count without physically replicating the book elements.

6 Query Optimization

Example 4 *Consider once more the query shown in figure 3. Even though the query is very simple, there are different ways to evaluate it. One join plan is to consider the join between* faculty *node and* secretary *node first, that is to find all the faculty members with a secretary, then, join the result with* RA *node. An alternative plan is to join* faculty *node with* RA *node first, then, join the result with* secretary *node. There are other possibilities too – one could join* RA *with* secretary *first. Such a join would result in a cartesian product if we were performing value-based joins, as in relational systems. Since these are structural joins, there is a "join predicate" between the two nodes of a "sibling" relationship, and such joins are quite reasonable to consider.*[5]

Query patterns that consist of several nodes have many more evaluation plans. The number of alternatives grows in factorial order. There can be orders of magnitude difference in the evaluation costs of different plans. A query optimizer must enumerate all (or a promising subset of) the evaluation plans, estimate their costs, and choose the one with lowest estimated cost to evaluate. Some initial work in this direction for XML query processing has been described in [36]. However this work considered only a very limited set of choices, and focused on navigational access methods, which we now know not to prefer.

6.1 Structural Join Order Selection

Join order selection is among the more important tasks of a relational query optimizer. Correspondingly, in an XML database, structural joins predominate. Every pattern match is computed as a sequence of structural joins, and the order in which these are computed makes a substantial difference to the cost of query evaluation.

In relational query processing it is almost always a good idea to evaluate selections first. In our case, we have an additional complication that it is not always a good idea to push selection predicates in all the way. This is for two reasons: structural join predicates may sometimes be more selective than a value-based selection predicate. Also, structural joins can be computed with node identifiers alone, whereas selection predicate evaluation may require access

[5] We do not, at present, have access methods for sibling structural joins implemented in Timber, though we are studying this issue. As such, the remainder of this section will restrict itself to consideration of parent-child and ancestor-descendant structural joins.

to the actual data (when an index is not present on the precise selection predicate). Consequently, (structural) join order optimization should also consider selection predicates being interspersed, rather than necessarily applied first. This further increases the size of solution space to be searched.

An adaptation of the standard dynamic programming techniques to our problem is as follows: We define *status* to be a reformed pattern-tree, with some sub-patterns in the pattern tree joined. Each of these joined sub-patterns coalesces into one node in the status, and the edges within these sub-patterns disappear. A *move*, which represents a join operation based on a single edge, transform one status into another. A cost value is associated with each move, based on the cardinalities of the nodes that participate in the join and the result size of the join. The starting status is exactly the pattern-tree itself, with an additional node created for each selection predicate. The additional node is attached as a child of the node the predicate references, and as a child of the least common ancestor of the nodes involved when multiple nodes are references. The status with only one node, which contains all the nodes in the original pattern-tree, is the final status. A sequence of moves that transform the starting status to a final status and has the minimum cost of all move sequences that can perform the same transformation is what we are looking for. We have a dynamic programming formulation, and can now solve this.

A difficulty with the preceding dynamic programming formulation is that the number of statuses (states) to be explored can be exponential in the size of the query pattern, making a full dynamic programming solution prohibitive. A less expensive solution can be developed based on the following observation:

By choosing an appropriate structural join algorithm, the results of a structural join can be output ordered by either of the two nodes involved in the join. No extra sorting is needed, and no blocking points created in the pipeline, if the *OrderBy* node in one join is a node involved in the next join. This leads to the following:

Theorem 1 *Any XML pattern match can be evaluated with a fully-pipelined evaluation plan to produce results ordered by any node in the pattern tree.*

Proof Sketch: Prove by induction on n, the total number of edges in a pattern. For the base case, the theorem obviously goes through for a query pattern with a single node and zero edges. For the inductive case, we can show that there is at least one pipelined plan, whose last join involves a sub-pattern which contains the result *OrderBy* node r, and a sub-pattern which contains one of its neighbors u. Each of these sub-patterns has less than n edges. By the inductive assumption, there is a pipelined plan for the first sub-pattern with results ordered by r and a pipelined plan for the second sub-pattern with results ordered by the neighbor node u. □

For full details on this topic, please see [56].

```
Algorithm FP_Optimization (PatternTree)
// Inputs: The query pattern to be evaluated,
// Output: Processing tree to evaluate the query.
minCost = ∞
For (each node V_i in PatternTree)
    cost = FP_OrderBy(PatternTree, V_i, joinplan)
    if (cost < minCost)
        minCost = cost; JoinPlan = joinplan;
output(JoinPlan);

Algorithm FP_OrderBy(PtTree, OdNode, JoinPlan)
// Inputs: PtTree: The query pattern to be evaluated;
//      OdNode: The node to be ordered by in output.
// Output: Processing tree and its cost
Neighbors = neighbors of OdNode
For each Neighbors[i]
    subpattern = SubPattern(PtTree,Neighbors[i],OdNode)
    cost = FP_OrderBy(subpattern, Neighbors[i], subplan);
Enumerate all the possible permutations of
subpatterns to join with OdNode.
Store the best plan in JoinPlan.
Store its cost in cost.
output(cost);

note:
Function SubPattern(PatterTree, Node1, Node2)
partitions the PatternTree by cutting the edge
between Node1 and Node2, and returns the
subpattern contains Node1.
```

Fig. 12 Algorithm *FP_Optimization* for Finding the Best Fully-Pipelined Evaluation Plan

6.2 Result Size Estimation

Query optimization enumerates a subset of all the possible join plans and picks the one with the lowest cost to execute. To estimate this cost, we need an accurate estimate of the cardinality of the final query result as well as each intermediate result for each query plan. Result size estimation is also useful for its own sake, in an Internet context, to provide users with quick feedback about expected result size before evaluating the full query result. There have been several recent studies of this topic, such as [1, 12]. We describe here the techniques incorporated in Timber that improve upon these previous works. Full details can be found in [54, 55].

Example 5 *Consider a simple pattern with only two nodes,* faculty *and* TA*, with parent-child relationship among them.*

There are three faculty *nodes and five* TA *nodes in the XML document. The schema says that a* faculty *can have any number of* TA*s. Without any further schema information, the best we can do in estimating the result size is to compute the product of the cardinality of these two nodes, which yields 15. Consider the fact that* faculty *nodes are not nested, one* TA *can only be the child of one* faculty *node, we can tell that the upper-bound of the result number is the cardinality of* TA *nodes, which is 5. But as we can see from the figure, the real result size is 2. The bias in the estimation is due to the fact that the structure information of the XML document is not caught.*

Position Histogram: Recall that a numeric Start and End label is associated with each data node in the database (XML document), defining a corresponding interval between these labels and the descendant nodes has an interval that is strictly included in its ancestor's interval. Taking the Start and End pair of values associated with each node that satisfy a predicate, we construct a two-dimensional histogram. Each grid cell in this *position histogram* represents a range of Start position values and a range of End position values. The histogram maintains a count of the number of nodes satisfying the predicate that have Start and End position within the specified ranges.

Each data node is mapped to a point in two-dimensional space. Node A is an ancestor of node B iff the start position of A is less than the start position of node B, and the End position of A is no less than the End position of node B. In other word, node A is to the left of and above node B in the position histogram. Therefore, given the position histogram of two node predicate, the estimate of the join result of this two nodes can be computed by looping through each grid cell in the histogram of one node predicate and counting the number of nodes (in the other histogram) which can have the desired relationship with a node in that grid cell. The estimate can be represented in forms of a position histogram itself, which makes it possible to estimate the result size for complex query patterns.

faculty

0	1
2	

TA

0	3
2	

Fig. 13 Position Histograms: The X-axis depicts start position value and Y-axis end position value. the

Let's have a look at the example XML document in figure 1 again. Consider a pattern tree with only two nodes, faculty and TA, with parent-child relationship among them. The 2×2 histograms of predicates "element tag = faculty" and "element tag = TA" are shown in figure 13. There are 55 nodes in the database. The left column in these histograms corresponds to elements that start in the first half of these nodes. The bottom row corresponds to elements that end in the first half. Thus, the first histogram says that there are 2 faculty in the first half of the database and one in the second half. The 0 in the top left indicates that there is no faculty element that has a large span starting in the first half and stretching through to the second half. The bottom right cell is empty because it can never be possible for an element to start in the second half and end in the first half.

Since TAs in the second half cannot work for faculty in the first half, and vice versa, we can now upper bound the number of matches to $(2 \times 2 + 1 \times 3) = 7$ instead of $(3 \times 5) = 15$. Note that the position histograms we used here is rather coarse, at 2×2. By refining the histogram, we can get a more accurate estimate.

7 Updates

The TAX tree algebra supports updates. We also have implemented access methods that support updates. However, XQuery does not (yet) support updates. There is a nice proposal [50] and separately an implemented Xupdate language [33]. Once a standard begins to emerge, it should (we hope) be straightforward for us to implement a parser for it and thus support it in Timber.

Beyond the simple implementation of updates, they do cause significant changes in the way the entire system is implemented. Since we do not use or enforce schema (or DTD) conformance, we are able to side-step many of the issues with respect to XML updates raised in [50]. However, our design is fundamentally that of a dynamic, rather than a static, database. All our indices are dynamic, and the underlying storage manager expects to manage the insertion and deletion of data blocks.

The start and end labels become an issue. Our current scheme is to use floating point numbers rather than integers, in effect leaving "holes" in the numbering, when initially assigning labels, with the hope that a moderate degree of modification can be absorbed within these holes. If there is an extremely localized sequence of inserts, these holes will not suffice. When such a situation arises, we start over and renumber every node, and reflect the new numbers at every place these may be used. This renumbering is a heavy-weight operation, which we hope will not be necessary too often. A more limited renumbering of some local region may suffice in some cases, but we have not yet worked out all the details of such a scheme.

```
FOR $a IN document("mbench.xml")//
                eNest[@aFour="0"]
FOR $b IN $a//eNest[@aSixteen="1"]
FOR $d IN $a//eNest[@aSixteen="2"]
FOR $f IN $a//eNest[@aSixteen="3"]
WHERE $b/eNest/@aSixtyFour="2"
  AND $d/eNest/@aSixtyFour="3"
  AND $f/eNest/@aSixtyFour="9"
RETURN
   <result>
      <A>{$a/text()}
           <B>{$b/text()}</B>
           <D>{$d/text()}</D>
           <F>{$f/text()}</F>
      </A>
   </result>
```

Table 1 XQuery statement run against the mBench data set

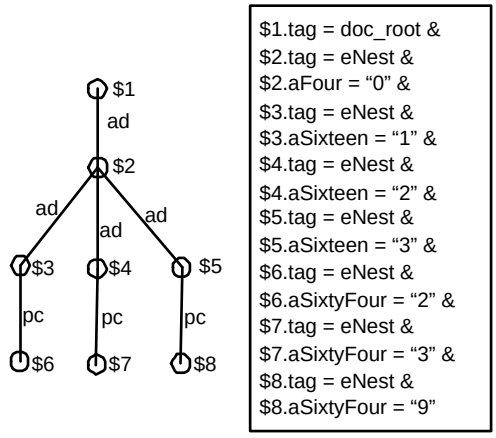

Fig. 14 The pattern tree generated from the XQuery statement of table 1. (Use as both selection tree $P1$ and projection tree $P2$).

Our initial insertion of elements has them ordered by start position. Changes in the sizes and numbers of elements in some range could cause pages to overflow or underflow. This is an intrinsically difficult problem. It is not acceptable to leave expansion room, as in the case of label values, since expansion room has an explicit cost in terms of wasted space and more disk pages to be accessed to obtain the same information. Heuristics are used in relational systems to maintain approximate clustering/ordering in the face of updates. We expect to use similar heuristics in Timber eventually. For the present, we let Shore use its default strategies to manage space for updates.

8 System Study

The Timber system is working for the most part, and we hope to be able to make a public release shortly. In this section, we work through one simple example in some detail to illustrate the operation of the system, and give some indication of performance.

Consider the query in table 1 against the mBench [66] 0.1x data set. The pattern tree produced by this query is shown in figure 14. This query would generate a selection and a projection. The selection $\sigma_{P1,SL}$ is applied on the database, with pattern tree $P1$ as in figure 14 and selection list $SL = \emptyset$. The projection $\pi_{P2,PL}$ is applied on the outcome of the selection, with pattern tree $P2$ as in figure 14, and projection list $PL = \$1, \$2, \$3, \$4, \$5$. The TAX expression corresponding to this query is as follows.

$$\pi_{P1,PL}\{\sigma_{P2,SL}[(\mathsf{mbench.xml})]\}$$

The RETURN part of the XQuery expression merely constructs a result tree by tying together nodes that have already been identified. In Timber, this construction is performed procedurally – since the output has to be produced any way, and since the manipulation is one-to-one, there is no benefit to set-oriented declarative processing of this part of the query. Note that the input to the algebra expression is a set of trees, in this case (as is frequently true for XML queries) a singleton set. The result of the selection is a set of trees, with cardinality considerably greater than one, that satisfy the given complex predicate.

Note that the selection and projection operators use identical pattern trees. We are able to exploit the reuse of pattern tree in the physical algebra. Our task then becomes to compute the sequence of structural joins and selections that comprise this pattern tree, and return the bindings for tuples of nodes that satisfy all the structural and node predicates. These node bindings are then used first in a selection operator, and then in a projection.

Note that in the RETURN clause, the text of the elements queried is required. Getting the text requires access to the database, which is expensive and therefore is postponed until the end of the evaluation. Hence, the plans produced by the optimizer will deal with indices only and node ids.

Our first step is to identify which of the predicates are indexed, and to determine the selectivity of each. Each of the attribute predicates is indexed, as shown in Table 2, and predicates on attribute aSixtyFour are more selective than the other attributes. The last column is the variable name given to the predicate. For the sake of simplicity, we will refer to the first predicate in the table as predicate A, and to the second predicate as B and so on.

Many query plans can be generated to evaluate this expression. In practice, the optimizer would go through a sig-

Ancs/Parent	Desc/Child	Naive Estimate	Desc Number	Parent-Child Join		Ancs-Desc Join	
				Histogram Estimate	Real Result	Histogram Estimate	Real Result
A	B	70,567,805	4,235	N/A	N/A	22,090	19,235
A	D	69,784,644	4,188	N/A	N/A	21,883	18,926
A	F	69,201,439	4,153	N/A	N/A	21,542	18,792
B	C	4,412,870	1,042	69	58	3,271	930
D	E	4,363,896	1,042	65	51	3,069	2,334
F	G	4,327,426	1,042	85	78	3,396	807

Table 3 Result Size Estimation for Pair Joins on the mBench Data Set. The first two columns are the predicates on the nodes being joined; the naive estimate is simply the product of the number of nodes participating in the join; the "Desc Number" uses schema information to upper bound the estimate to the number of descendant nodes; the "Histogram Estimate" uses 10×10 position histograms. Estimates are shown for parent-child and ancs-desc joins where available.

Predicate	Node Count	Variable
eNest[@aFour="0"]	16,663	A
eNest[@aSixteen="1"]	4,235	B
eNest[@aSixteen="2"]	4,188	D
eNest[@aSixteen="3"]	4,153	F
eNest[@aSixtyFour="2"]	1,042	C
eNest[@aSixtyFour="3"]	1,042	E
eNest[@aSixtyFour="9"]	1,042	G

Table 2 Characteristics of Some Predicates on the mBench Data Set

nificant search to find the plan with least estimated cost. For the current discussion, we restrict ourselves to five possible choices of plans for pattern tree evaluation presented in figure 15. In all cases, the actual access to data, and getting the text of the elements is deferred.

The estimated intermediate result sizes are shown in Table 3. There can be an eNest node under another eNest node in the database. This is called the *overlap property*. Note that the estimate exploiting this minimal amount of schema information, in columns 5 and 7, is in each case very close to the real result, shown in columns 6 and 8. As we can see from columns 7 and 8 rows 1 through 3, using position-histogram can accurately estimate the result size for ancs-desc structural joins such as A and B join. However, ancs-desc structural joins such as B and C, the estimate is not very good (columns 7 and 8 rows 4 and 6). The reason for this is that the ancs-desc estimate is obtained using position histograms which are not good when the parent node has the overlap property. But when using leveled histograms to obtain the parent-child estimates for the same structural joins, the results are accurate as shown in columns 5 and 6 rows 4 through 6.

There are approximately 130,000 nodes in the mBench 0.1x data set. Predicates C, E, and G are the most selective of the predicates. Unfortunately, the three cannot be

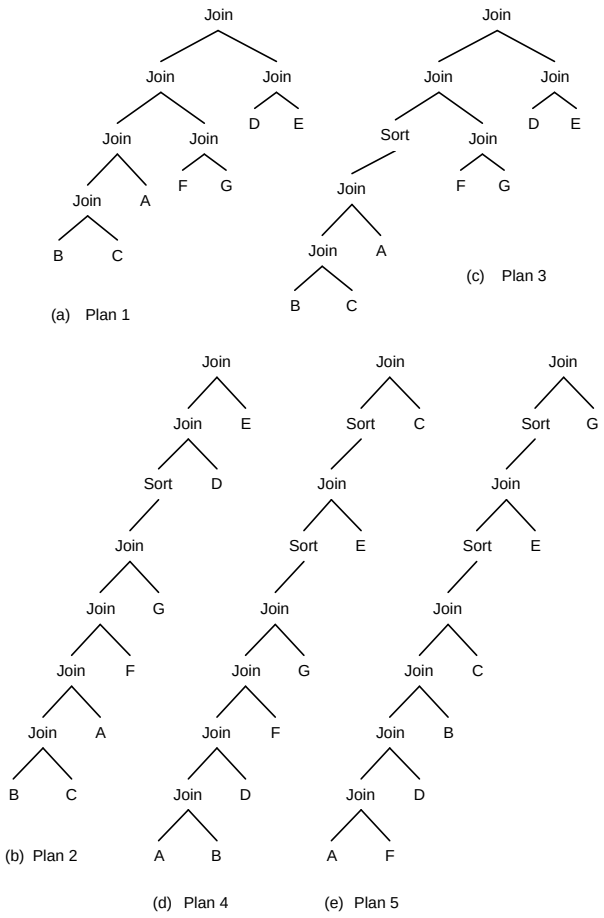

Fig. 15 Five Alternative Query Plans: Each leaf is an index access, each internal node is a stack-based structural join ordered by ancestor or descendant (as needed); where sorting is required, this is shown explicitly.

combined directly, and require A, B, D, and F as common parent/ancs nodes for this purpose.

Fig 15 shows five alternative plans. The optimizer chooses Plan 1 as the optimal plan based on these estimates. Notice that this plan is bushy and non-blocking. The best left deep plan, such as Plan 2, gets into a blocking situation, where we need the results sorted by A for the next join, with D, but the results are available sorted by F. An option is to sort earlier such as Plan 3. This gives a plan that is comparable to plan 1. We show Plans 4 and 5, just to make the (obvious) point that left-deep plans can be really bad.

The actual execution times for these plans and several more are shown in Table 4. These experiments were run on a standard desk-top IBM compatible PC, running Windows NT Workstation v4.0. The machine had a single 500 MHz Intel Pentium III CPU and 256 MB of memory. Furthermore, we restricted Shore to use only 32MB of memory for its buffer pool, in 8KB pages. Each test was run 5 times. The highest and lowest numbers were discarded and the average of the middle three was reported. The time to parse and optimize the query is small in comparison, and is not included in the times reported in Table 4.

Plan Number	Plan Description	Running Time
1	Optimal Bushy	1.4 sec.
2	Blocking Left-Deep	193.5 sec.
3	Blocking Bushy	1.532 sec.
4	Double-blocking Left-Deep	414.7 sec.
5	Double-blocking Left-Deep	413.4 sec.
6	No Pattern Tree Reuse	2.9 sec.
7	Navigational	2,121.3 sec.

Table 4 Performance of Different Query Plans

In addition to the five query plans described above, in Table 4 we also present performance results for two other plans, which are expected to perform poorly. Plan 6 shows the impact of not reusing the pattern tree computation. Each use of the pattern tree is conducted according to the best plan – Plan 1. The cost is expected to be about twice. Plan 7 uses no indices at all. It scans the document for A nodes and for each, it scans its subtree for B, D, and F nodes. And for each of the latter nodes, it scans their subtrees for the rest of the pattern. This is probably the naive approach that one thinks of first, when given this problem. The poor performance of these two additional plans validates many of the design choices we have described above.

9 Conclusion

We have described the architecture and overall design of the Timber native XML database system currently being implemented at the University of Michigan. Through the use of a carefully designed tree algebra, as well as the judicious use of novel access methods and optimization techniques, we have created the foundation for a high performance database system capable of operating at large scale.

The system has been designed in a modular fashion, with an overall architecture as similar to a relational database as possible. We have attempted to reuse as much standard technology as possible. Thus, standard value-based hash and B-tree indices can be used with only small changes. Similarly, transaction management is largely unchanged, and in our system is implemented by Shore. Consequently, little effort is required to move Timber from a single-user system to a multi-user system.

We already have a few potential users of Timber. We expect, in the coming months, to benefit from their experiences with the system, and to expand its facilities accordingly. We also expect to make a public release of Timber software.

References

1. Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. VLDB Conf.*, Rome, Italy, 2001.
2. Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE Conf.*, Mar. 2002.
3. Shurug Al-Khalifa and H. V. Jagadish. Multi-level Operator Combination in XML Query Processing. In *Proc. CIKM Conf.*, Nov. 2002.
4. Apache Web Site. Item 10 on Xindice FAQ. `http://xml.apache.org/xindice/FAQ` .
5. D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, and P. Rodriguez-Gianolli. ToX - The Toronto XML Engine. *Proc. Intl. Workshop on Information Integration on the Web*, Rio de Janeiro, 2001.
6. C. Baru, A. Gupta, B. Ludaescher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. In *Exhibitions Program of SIGMOD Conf.*, 1999.
7. E. Bertino. An Indexing Technique for Object-Oriented Databases. In *Proc. ICDE Conf.*, pages 14–22, 1991.
8. R.S. Boyer, J. S. Moore. A Fast String Searching Algorithm. Communications of the Association for Computing Machinery, 20(10), 1977, pp.762-772.

9. M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up Persistent Applications. In *Proc SIGMOD Conf.*, pages 383–394, 1994.
10. D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A Query Language for XML. W3C Working Draft. Available from `http://www.w3.org/TR/xquery`
11. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Informal Proc. WebDB Workshop*, pp. 53–62, May 2000.
12. Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting Twig Matches in a Tree. In *Proc. ICDE*, Heidelberg, Germany, pp. 595–604, Mar. 2001.
13. Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, Donghui Zhang: Efficient Complex Query Support for Multiversion XML Documents. In *Proc. EDBT*, Prague, Czech Republic, pp. 161–178, 2002.
14. Edith Cohen, Haim Kaplan, Tova Milo: Labeling Dynamic XML Trees. In *Proc. PODS Conf.*, pages 271-281, June 2002.
15. Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. *Proc. VLDB Conf.*, pages 341–350, Sep. 2001.
16. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. Submission to the World Wide Web Consortium 19-August-1998. Available from `http://www.w3.org/TR/NOTE-xml-ql`, 1998.
17. L. Fegaras and R. Elmasri. Query Engines for Web-Accessible XML Data. In *Proc. VLDB Conf.*, Rome, Italy, September 2001.
18. Mary F. Fernandez, Jerome Simeon, Philip Wadler. An Algebra for XML Query. *Proc. FSTTCS Conf.*, pages 11-45, 2000.
19. Thorsten Fiebig, Guido Moerkotte: Evaluating Queries on Structure with eXtended Access Support Relations. *Informal Proc. WebDB Workshop*, pages 125–136, 2000.
20. D. Florescu, G. Graefe, G. Moerkotte, H. Pirahesh, and H. Schning. Panel: XML Data Management: Go Native or Spruce up Relational Systems? In *Proc. SIGMOD Conf.*, Santa Barbara, California, May 2001.
21. D. Florescu and D. Kossman. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
22. Leonidas Galanis, Efstratios Viglas, David J. DeWitt, Jeffrey. F. Naughton, and David Maier. Following the Paths of XML Data: An Algebraic Framework for XML Query Evaluation. University of Wisconsin Tech. Report.
23. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB Conf.*, pages 436-445, Athens, Greece, August 1997.
24. R. Goldman and J. Widom. Summarizing and Searching Sequential Semistructured Sources. Technical Report, March 2000.
25. H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL Conf.*, Rome, Italy, Sep. 2001.
26. Carl-Christian Kanne, Guido Moerkotte: Efficient Storage of XML Data. Poster abstract in *Proc. ICDE Conf.*, page 198, San Diego, CA, March 2000.
27. C. C. Kanne and G. Moerkotte. Efficient Relational Storage and Retrieval of XML Documents. Technical Report 8/99, University of Mannheim, 1999.
28. Dao Dinh Kha, Masatoshi Yoshikawa, Shunsuke Uemura: An XML Indexing Structure with Relative Region Coordinate. In *Proc. ICDE Conf.*, pages 313-320, 2001.
29. C. Kilger and G. Moerkotte. Indexing Multiple Sets. *Proc. VLDB Conf.*, pages 180–191, Sept. 1994.
30. M. Klettke, H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *Informal Proc. WebDB Workshop*, pages 151–170, 2000.
31. D. E. Knuth, J. H. Morris (Jr) and V. R. Pratt, 1977, Fast pattern matching in strings, SIAM Journal on Computing 6(1):323-350.
32. S. A. T. Lahiri and J. Widom. Ozone: Integrating Structured and Semistructured Data. In *Proc. DBPL Conf.*, Kinloch Rannoch, Scotland, Sep. 1999.
33. Andreas Laux and Lars Martin. XUpdate Working Draft, Sep. 2000. Available at `http://www.xmldb.org/xupdate/xupdate-wd.html`.
34. Pedro José Marrón and Georg Lausen. On Processing XML in LDAP. *VLDB*, pages 601—610, 2001.
35. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management Systems for Semistructured Data. *SIGMOD Record 26(3)*, pages 54–66, 1997.
36. J. McHugh and J. Widom. Query Optimization for XML. In *Proc. VLDB Conf.*, pages 315–326, 1999.
37. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Technical Report, January 1998. Available at `http://www-db.stanford.edu/lore/pubs`.
38. Microsoft XQuery Language Demo. Online at `http://131.107.228.20/xquerydemo/`
39. Jeffrey Naughton, David DeWitt, David Maier, et al. The Niagara Internet Query System. Available at `http://www.cs.wisc.edu/niagara/papers/NIAGARAVLDB00.v4.pdf`
40. U. of Wisconsin. The Niagara system. Available from `http://www.cs.wisc.edu/niagara/`.
41. S. Paparizos, S. Al-Khalifa, H.V. Jagadish, L. Lakshmanan, A. Nierman, D. Srivastava and Y. Wu. Grouping in XML. In EDBT 2002 Workshop on XML-Based Data Management (XMLDM'02).

42. D. Quass, J. Widom, R. Goldman, H. K, Q. Luo, J. Mchugh, A. Rajaraman, H. Rivero, S. . Abiteboul, J. Ullman, and J. Wiener. Lore: A Lightweight Object Repository for Semistructured Data. *Proc. SIGMOD Conf.*, page 549, 1996.
43. J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.
44. Kanda Runapongsa and Jignesh M. Patel. Storing and Querying XML Data in ORDBMSs. *EDBT XML-Based Data Management (XMLDB) Workshop*, March 24, 2002, Prague, Czech Republic.
45. Arnaud Sahuguet. Kweelt: More Than Just "Yet Another Framework to Query XML!". *Proc. SIGMOD Conf.*, Santa Barbara, CA, 2001. Software available from `http://db.cis.upenn.edu/Kweelt/`.
46. Harald Schoning. Tamino - A DBMS designed for XML. In Proc. ICDE Conf., pp. 149–154, 2001.
47. Harald Schoning and J. Wasch. Tamino - An Internet Database System. In Proc. EDBT Conf., pp. 383–387, 2000.
48. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.DeWitt, and J.Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. VLDB Conf.* pages 302–314, Edinburgh, Scotland, Sep. 1999.
49. T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Proc. DEXA Conf.*, pages 206–217, Florence, Italy, September 1999.
50. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. SIGMOD Conf*, 2001.
51. Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. The Design and Performance Evaluation of Various XML Storage Strategies. University of Wisconsin Technical Report, 2000.
52. U. of Washington. The Tukwila system. Available at `http://data.cs.washington.edu/integration/tukwila/`.
53. Bennet Vance and David Maier. Rapid Bushy Join-order Optimization with Cartesian Products. In Proc. SIGMOD Conf., pages 35–46, Montreal, Quebec, 1996.
54. Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proc. EDBT Conf.*, Prague, Czech Republic, Mar. 2002.
55. Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Using Histograms to Estimate Answer Sizes for XML Queries. *Information Systems*, to appear, 2002.
56. Yuqing Wu and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization In *Proc. ICDE Conf.*, Bangalore, India, Mar. 2003.
57. W3C DOM Working Group. Document Object Model. Available at `http://www.w3.org/DOM/`.
58. W3C. Extensible Markup Language (XML) 1.0. W3C Recommendation. Available at `http://www.w3.org/XML`.
59. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management systems. In *Proc. SIGMOD Conf.*, Santa Barbara, CA, 2001.
60. Tamino Developer Community QuiP, a W3C XQuery Prototype. Available at `http://www.softwareag.com/developer/quip`.
61. eXcelon Corp. eXcelon XML platform. Available at `http://www.exceloncorp.com/platform/extinfserver.shtml`.
62. X-Hive Corp. X-Hive/DB. Available at `http://www.x-hive.com`.
63. dbXML Group. dbXML Core. Available at `http://www.dbxml.org`.
64. W3C. XML Schema. W3C Recommendation. Available at `http://www.w3.org/XML/Schema`.
65. OASIS Technical Committee. Relax NG. Available at `http://www.oasis-open.org/committees/relax-ng/`.
66. The Michigan Benchmark Team. The University of Michigan, Available at `http://www.eecs.umich.edu/db/mbench`.
67. The Timber Team. The University of Michigan, Available at `http://www.eecs.umich.edu/db/timber`.