Querying XML in TIMBER

Yuqing Wu
Indiana University
yuqwu@indiana.edu

Stelios Paparizos
Microsoft Research
steliosp@microsoft.com

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

Abstract

In this paper, we describe the TIMBER *XML database system implemented at University of Michigan.* TIMBER *was one of the first native XML database systems, designed from the ground up to store and query semi-structured data. A distinctive principle of* TIMBER *is its algebraic underpinning. Central contributions of the* TIMBER *project include: (1) tree algebras that capture the structural nature of XML queries; (2) the stack-based family of algorithms to evaluate structural joins; (3) new rule-based query optimization techniques that take care of the heterogeneous nature of the intermediate results and take the schema information into consideration; (4) cost-based query optimization techniques and summary structures for result cardinality estimation; and (5) a family of structural indices for more efficient query evaluation. In this paper, we describe not only the architecture of* TIMBER, *its storage model, and engineering choices we made, but also present in hindsight, our retrospective on what went well and not so well with our design and engineering choices.*

The TIMBER system [10, 16] was developed at the University of Michigan, Ann Arbor, beginning 1999. It was an early native XML data management system. In this retrospective, we take stock of our work over the past nine years. Figure 1 provides an overview of the major system components. Secs. 1 through 4 describe the underlying algebra, query evaluation methods, query optimization, and indices, respectively. Sec. 5 mentions aspects of TIMBER not included in this article. Sec. 6 concludes with a retrospective view.

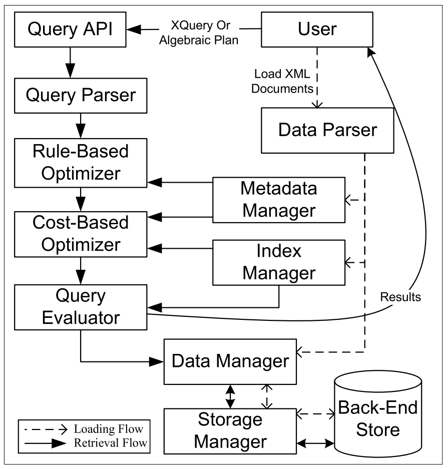


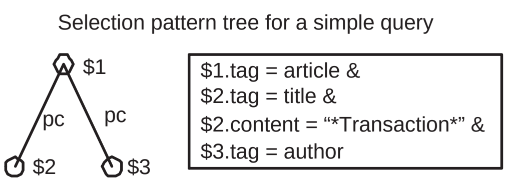
Figure 1: TIMBER Architecture: XML documents are parsed and nodes stored individually in the back-end store. Parsed queries, from multiple supported interfaces, go through a query optimizer to the query evaluator in a relatively standard overall database system architecture.

1 Algebra

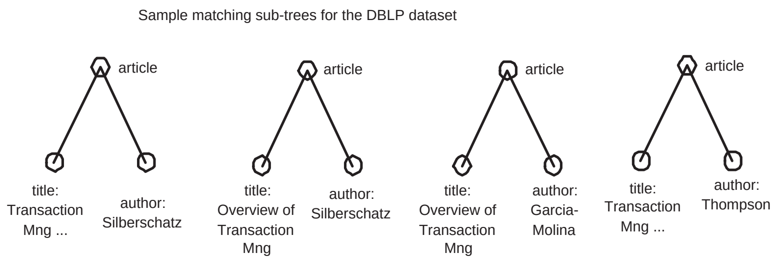
Relational algebra has been a crucial foundation for relational database systems, and has played a large role in enabling their success. A corresponding XML algebra for XML query processing has been more elusive, due to the comparative complexity of XML, and its history.

In the relational model, a tuple is the basic unit of operation and a relation is a set of tuples. In XML, a database is often described as a forest of rooted node-labeled trees. Hence, for the basic unit and central construct of our algebra, we chose an *XML query pattern* (or

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering


Selection pattern tree for a simple query
$1
pc
pc
$2
$3
$1.tag = article &
$2.tag = title &
$2.content = "*Transaction*" &
$3.tag = author

(a) Pattern tree for 'Select articles with some author and with title that contains Transaction'.


Sample matching sub-trees for the DBLP dataset
article
title: Transaction Mng ...
author: Silberschatz
article
title: Overview of Transaction Mng
author: Silberschatz
article
title: Overview of Transaction Mng
author: Garcia-Molina
article
title: Transaction Mng ...
author: Thompson

(b) Witness trees from the matching of the tree in Figure 2(a) to DBLP.

Figure 2: Pattern Tree and Witness Trees.

twig), which is represented as a rooted node-labeled tree. An example of such tree, we call it *pattern tree*, is shown in Figure 2(a). An edge in such tree represents a structural containment relationship, between the elements represented by the respective pattern tree nodes. The containment relationship can be specified to be either immediate (parent-child relationship) or of arbitrary depth (ancestor-descendant relationship). Nodes in the pattern tree usually have associated conditions on tag names or content values.

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

Using this basic primitives, we developed an algebra, called *Tree Algebra for XML* (TAX) [12], for manipulating XML data modeled as forests of labeled ordered trees. Motivated both by aesthetic considerations of intuitiveness, and by efficient computability and amenability to optimization, we developed TAX as a natural extension of relational algebra, with a small set of operators. TAX is complete for relational algebra extended with aggregation, and can express most queries expressible in popular XML query languages.

Ordering and Duplicates XML itself incorporates semantics in the order in which the data is specified. XML queries have to respect that and produce results based on this *document order*. XQuery takes this concept even further and adds an extra implicit ordering requirement. The order of the generated output is sensitive to the order the variable binding occurred in the query, the *binding order*. Additionally, a FLWOR statement in XQuery may include an explicit *ORDERBY* clause, specifying the ordering of the output based on the value of some expressions – this is similar in concept to ordering in the relational world and SQL.

Although XML and XQuery require ordering, many "database-style" applications could not care less about order. This leaves the query processing engine designer in a quandary: should order be maintained, as required by the semantics, irrespective of the additional cost; or can order be ignored for performance reasons. What we would like is an engine where we pay the cost to maintain order when we need it, and do not incur this overhead when it is not necessary. In algebraic terms, the question we ask is whether we are manipulating sets, which do not establish order among their elements, or manipulating sequences, which do.

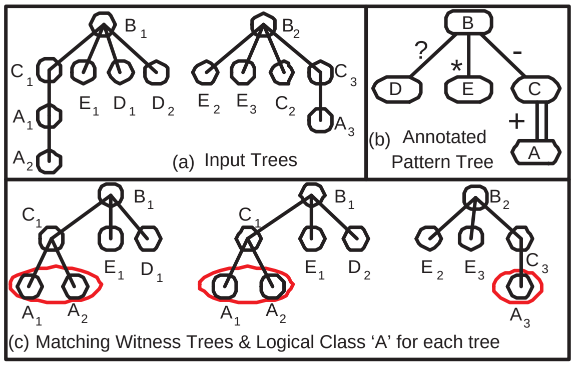
The solution we proposed is to define a new generic *Hybrid Collection* type, which could be a set or a sequence or even something else. We associate with each collection an *Ordering Specification O-Spec* that indicates precisely what type of order, if any, is to be maintained in this collection.

Duplicates in collections are also a topic of interest, not just for XML, but for relational data as well. The more complex structure of XML data raises more questions of what is equality and what is a duplicate. Therefore there is room for more options than just sets and multi-sets. Our solution is to extend the *Hybrid Collection* type with an explicit *Duplicate Specification D-Spec*. Using our Hybrid Collections we extended our algebra [18] thus we were able to develop query plans that maintain as little order as possible during query execution, while producing the correct query results and managing to optimize duplicate elimination steps.

Tree Logical Classes (TLC) for XML XQuery semantics frequently requires that nodes be clustered based on the presence of specified structural relationships. For example the RETURN clause requires the complete subtree rooted at each qualifying node. A traditional pattern tree match returns a set of *flat* witness trees satisfying the pattern, thus requiring a succeeding grouping step on the parent (or root) node. Additionally, in tree algebras, each algebraic operator typically performs its own pattern tree match, redoing the same selection time and time again. Intermediate results may lose track of previous pattern matching information and can no longer identify data nodes that match to a specific pattern tree node in an earlier operation. This redundant work is unavoidable for operators that require a homogeneous set as their input without the means for that procedure to persist.

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

Annotated Pattern Trees accept edge matching specifications that can lift the restriction of the traditional one-to-one relationship between pattern tree node and witness tree node. These specifications can be "-" (exactly one), "?" (zero or one), "+" (one or more) and "*" (zero or more). Figure 3 shows the example match for an annotated pattern tree. Once the pattern tree match has occurred we must have a logical method to access the matched nodes without having to reapply a pattern tree matching or navigate to them. For example, if we would like to evaluate a predicate on (some attribute of) the "A" node in Figure 3, how can we say precisely which node we mean? The solution to this problem is provided by our Logical Classes. Basically, each node in an annotated pattern tree is mapped to a set of matching nodes in *each* resulting witness tree – such set of nodes is called a *Logical Class.* For example in Figure 3, the gray circles indicate how the "A" nodes form a logical class for each witness tree. Using this techniques we extended TAX into our *Tree Logical Class* (TLC) algebra [19].



(a) Input Trees
(b) Annotated Pattern Tree
(c) Matching Witness Trees & Logical Class 'A' for each tree
B1
B2
C1
C2
C3
A1
A2
A3
E1
E2
E3
D1
D2
B
D
E
C
A
?
*
-
+

Figure 3: Sample Match for Annotated Pattern Tree

2 Query Evaluation

Data Storage The unit of storage in TIMBER is a node. For efficiency reasons, a node in the TIMBER *Data Manager* is not exactly the same as a DOM [22] node: there is a node corresponding to each element, with links to nodes corresponding to the first and last sub-elements; 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; the content of an element node, if any, is pulled out into a separate child node, in honor of interleaving of multiple sub-elements and text contents of mixed-type elements. We ignored all processing instructions and comments, which can be extended easily by creating nodes of those types.

In semi-structured data, the essential of the structural properties is reflected by the containment relationship between an element and its sub-elements. Establishing parent-child (or ancestor-descendant) relationships among nodes are the center parts of XML queries, and a sub-tree rooted at certain nodes are frequently demanded as query results. As such, the determination of the containment relationships is at the core of XML query processing. In TIMBER, we facilitated such operation by associating a numeric vector (start, end, level) with each data node in the database. The start and end labels of a node are the pre-order and post-order traversal of the node in the XML tree structure. Together, they define a corresponding interval such that every descendant node has an interval that is strictly contained in its ancestors' intervals. The level label reflects the depth of the node in the document for establishing parent-child relationships between nodes. Formally,

- A node $n_1(S_1, E_1, L_1)$ is the ancestor of node $n_2(S_2, E_2, L_2)$ iff $S_1 < S_2 \wedge E_1 > E_2$.
- A node $n_1(S_1, E_1, L_1)$ is the parent of node $n_2(S_2, E_2, L_2)$ iff $S_1 < S_2 \wedge E_1 > E_2 \wedge L_1 = L_2 - 1$.

The (start, end, level) vector of each node is generated automatically by the system during the data parsing process and stored together as attributes with each node. An additional doc label is associated with each node, such that the vector (doc, start, end, level) serves as a logical identifier for each node in a TIMBER database. We chose to store the nodes in the order of their start key, e.g. in document order, such that nodes within a sub-tree are always clustered together, hence guarantee efficient access of all nodes in a sub-tree, given the root.

Structural Join Evaluation TIMBER includes access methods corresponding to all operators in the TLC algebra. TLC (and TAX) operators have two parts: pattern match for witness tree identification followed by the actual operator application to the matched witness tree. Therefore, efficient pattern matching is crucial.

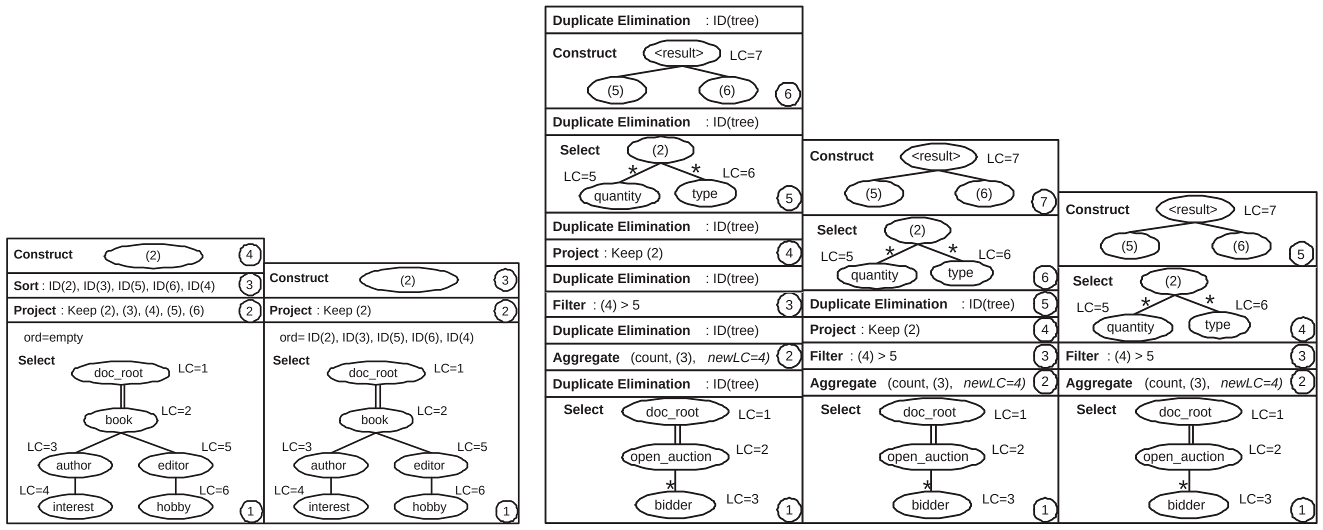
Pattern matching comprises two steps: first, value indices are used to look up matches for individual nodes in the pattern; then, *structural joins* are computed amongst these matched nodes. Structural join is the dominate operation in XML query evaluation both in terms of the frequency of usage and the cost. Consequently, efficient implementation of the structural join is critical to the efficient evaluation of XML queries in general.

Using the formulae of the containment relationship presented above, 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, as suggested in [2, 26]. We exploited the tree structure of XML to do better. We have developed, and used in TIMBER, a whole *Stack-Tree* family of structural join algorithms.

The basic idea of the *Stack-Tree* algorithm is to take the two input operand lists, AList and DList, both ordered by the start position and merge them using a stack. It takes advantage of the fact that in a depth-first traversal of the database tree, every ancestor-descendant pair appears on a stack with the ancestor below the descendant, and perform a limited depth-first traversal, skipping over nodes that are not in either input candidate list (AList or DList). The output is a list of matching pairs, which satisfy the designated structural relationship, in ascending order of the *start key* of either the ancestor or descendent participating in the join. The sort order of the output is very important for pipelined query evaluation. The *Stack-Tree* algorithm is a non-blocking algorithm and can produce result as the join happens.

Small variations of the algorithms described above can be used if the desired structural join is a parent-child join rather than an ancestor-descendant 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).

The algorithm requires an in-memory stack whose size is bounded by the maximum depth of the XML document. Even for the variant which requires the output to be sorted by the ancestor node, in which results has to be temperedly stored for each of the node in the stack, till 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 for any result page. 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). Experiments show that these algorithms far outperform the navigation-based join algorithms, as well as the RDB implementation, in all cases.



(a) On the right the rewritten plan having pushed the *Sort* into the *Select.*

(b) Minimizing Duplicate Elimination procedures.

Figure 4: Order and Duplicate Rewrites.

3 Query Optimization

3.1 Algebraic Rewrites

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

Grouping: While SQL allows for grouping operations to be specified explicitly, XQuery provides only implicit methods to write such queries. For example consider a query that seeks to output, for each `author` all the `titles` of `articles` he authored. A possible XQuery statement for this purpose (i.e. XQuery use case 1.1.9.4 Q4 from `http://www.w3.org/TR/xquery-use-cases`) would involve a nested FOR loop. A direct implementation of this query as written would involve two distinct retrievals from the bibliography database, one for `authors` and one for `articles`, followed by a join. Yet, one of our basic primitives of our algebra is a GROUPBY operator, thus enabling us to produce a smarter plan than the one dictated by XQuery. The power of the algebra allows for the transformation of the naïve join plan into a more efficient query plan using grouping – overcoming the XQuery nuances and making it similar to a relational query asking for the same information [17].

Duplicates and Ordering: As we discussed in Section 1, smart operation placement of ordering and duplicate elimination procedures can cause orders of magnitude difference in evaluation performance. We show two examples of such rewrites. Figure 4(a) shows how we optimize ordering. The rewrite takes advantage of our extended operations that use *Ordering Specification* annotations to push the `Sort` procedure into the original `Select`. Thus, the rewrite provides the cost-based optimizer with the means to efficiently plan the pattern tree match using the appropriate physical access methods, without having to satisfy a blocking `Sort` operation at the final step of the query plan. Figure 4(b) illustrates how the duplicate elimination procedures can be minimized. First, we naïvely force a duplicate elimination after every operation to produce the correct behavior. Then our technique detects and removes all redundant procedures by checking which operations will potentially produce duplicates. With the last step, we took advantage of our partial duplicate collections and manage to remove the duplicate elimination procedure completely. Details for both techniques can be found in [18].

3.2 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. What's different from the relational engine is that (1) in the context of XML structural relationships can be as selective as value predicates; and (2) with the help of value indices and the structural join algorithms, structural join can be evaluated without accessing the original data. Therefore, it is not always a good idea to push selection predicates all the way down.

We proposed a set of algorithms for selecting the optimal join order for computing a pattern match [25]. The Dynamic Programming (DP) algorithm is capable of enumerating all possible evaluation plans and estimating their costs. This guarantees that the DP algorithm can select the optimal evaluation plan. However, the number of plans explored can be exponential in the size of the query pattern, making a full dynamic programming solution prohibitive. The Dynamic Programming with Pruning (DPP) algorithm explore only the most promising plans by pruning the costly plan in the early stage of plan enumeration.

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 finding that *any XML pattern matching can be evaluated with a fully-pipelined evaluation plan to produce results ordered by any node in the pattern tree*.

Contrary to the common wisdom in RDB query evaluation that a left-deep plan usually outperform bushy plans, the optimal evaluation plan for XML pattern matching can be a bushy plan. The Fully-Pipelined (FP) algorithm explores only these index-only plans, left-deep or bushy, in the join order selection process. Our experiments showed that not only can the FP algorithm select a very good (close to optimal) evaluation plan, it is itself also much more efficient than the DP and DPP algorithms.

3.3 Result Size Estimation

Query optimization techniques, as presented above, 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. Even though the attributes participating in a join operation in RDB are often assumed to be independent, such assumption usually results into biased cardinality estimation in the context of XML, due to the fact that nodes *are* correlated, via parent-child or ancestor-descendant relationships that are natural to XML data.

The numeric start and end labels associated with each data node in the database define a corresponding interval between them. ignoreDescendant nodes have an interval that is strictly included. Taking the start and end pair of values associated with each node that satisfy a predicate, we constructed a two-dimensional histogram [23, 24]. 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 nodes satisfying the predicate that have start and end position within the specified ranges. Each data node is mapped to a point in this 2D space. Node A is an ancestor of node B iff node A is to the left of and above node B. Therefore, given the position histograms 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 sizes for complex query patterns.

4 Indexing

There is a rich history of work on index structures suited to specific purposes, in particular, the work done in the context of object-oriented systems, such as [4, 14], and more resent work on structural indices such as DataGuide [8] and A(k)-index [13]. More importantly, we drew inspiration from the theoretical work that studies the properties of the XPath language, and found that suitable indices for XML should be those that (1) are based on the partition of XML components which corresponds to the partition induced by important sub-language of

XPath; (2) label the partition to facilitate lookup; (3) organize the partitions in a way that facilitates retrieval of one or more such partitions; (4) support index-only plans for answering most XPath queries.

On the data side, seeing XML document D as a node-labeled tree, we formally define it as a 4-tuple (V, Ed, r, λ), with V the finite set of nodes, $Ed \subseteq V$ x V the set of edges, $r \in V$ the root, and $\lambda \colon V \to \mathcal{L}$ a node-labeling function into the set of labels $\mathcal{L}$. For a given pair of nodes m and n in an XML document D where m is an ancestor of n, we define its associated *label-path* to be the unique path between m and n, denoted $LP(m, n)$. Given a node n in D, and a number k, we define the *k-label-path* of n, denoted $LP(n, k)$, to be the label-path of the unique downward path of length l into n where $l = \min\{height(n), k\}$.[1].

We use the notion of label-paths to define $\mathcal{N}[k]$-equivalence such that two nodes are $\mathcal{N}[k]$-equivalent if the upward path of length k from them are identical. The *$\mathcal{N}[k]$-partition* of an XML document D is then defined as the partition induced by this equivalence relation. It immediately follows that each partition class C in the *$\mathcal{N}[k]$-partition* can be associated with a unique label-path, the label-path of the nodes in C, denoted $LP(C)$. On the other hand, a k-label-path p in an XML document D uniquely identifies an *$\mathcal{N}[k]$-partition* class, which we denote as $\mathcal{N}[k][p]$. In [7] we proved that the $\mathcal{N}[k]$-partition and the $\mathcal{A}[k]$-partition are the same. Similarly, we can define the *$\mathcal{P}[k]$-equivalent* relation between node pairs and the *$\mathcal{P}[k]$-partition* of node pairs in an XML document induced by the *$\mathcal{P}[k]$-equivalent* relationship.

XPath query language has been studied by many researchers. The XPath algebra, as proposed in [9], is defined as follows:

$$XPath\,algebra := \varepsilon|\emptyset|\downarrow|\uparrow|\ell|\lambda|E_1 \diamond E_2|E_1[E_2]|E_1 \cup E_2(X)|E_1 \cap E_2|E_1 - E_2$$

Where E_1 and E_2 are XPath algebra expressions. The *path semantics* of the algebra results into a set of node pairs, while the node semantics produces results in the form of node set. We focused our study on a few sub-algebras of XPath. The $\mathcal{D}$ algebra consists of the expressions in the XPath algebra without occurrences of the set operators, predicates ([]), or the $\uparrow$ primitive. The $\mathcal{D}^{[]}$ algebra consists of the $\mathcal{D}$ algebra plus predicates. More importantly, we studied a localized version of these sub-languages, e.g. $\mathcal{D}[k]$ and $\mathcal{D}^{[]}[k]$, restricting the length of the path to k.

The partition induced by a query language $\mathcal{F}$ under the path-semantics is defined as a partition of node pairs whereas two pairs are —em $\mathcal{F}$-equivalent to each other iff for any XML document D and any query expression in $f \in \mathcal{F}$, the node pairs are either together in $f(D)$, or together not in $f(D)$ [2].

We proved in [7] that the *$\mathcal{P}[k]$-partition* is the same as the *$\mathcal{D}[k]$-partition* of node pairs. In addition, we proved in [5] that every $\mathcal{D}^{[]}$ expression can be rewritten into sub-expressions in $\mathcal{D}[k]$, with the help of the inverse ($^{-1}$) operation, project operation and natural join operation to stitch the results of the sub-expressions together. Therefore, a proper index based on the *$\mathcal{P}[k]$-partition* of an XML document, with a modest k value, having the index entries featuring the (start, end, level) trio, is sufficient to support index-only evaluation plan for any XPath queries. Based on this theoretical result, we designed the $\mathcal{P}[k]$-Trie index, which uses the reversed label path as index key, and organizes the index entries in a trie structure. This index (1) has a reasonable size with a modest k; (2) is balanced, with k as the upper bound for the length of the search path; and (3) can answer queries of any length and with any arbitrary branching predicates with index-only plan. Our experiments showed that it outperformed the $\mathcal{A}[k]$-index by orders of magnitude.

5 Other Contributions

Space constraints prevent us from describing contributions of the TIMBER project beyond the core components discussed above. In this section, we briefly mention some of these other efforts.

XML holds out the promise of integrating unstructured text with structured data. The challenge lies in developing query mechanisms that can marry the very different IR-style queries appropriate for text with the

[1] $height(n)$ denotes the height of node n in D.

[2] Similarly, we can define the partition induced by $\mathcal{F}$ under the node-semantics.

structured representations of logic used by databases. The TIX algebra [3] was an early effort at bringing these two together.

Uncertainty in databases has recently become a hot topic. Uncertainty is particularly important in the context of XML because of the nature of applications where information may be obtained from sources that are less uniformly structured, less under our control, and less reliable. The ProtDB [15] facility in TIMBER provides a natural model to represent probabilistic data in XML, and to query it efficiently.

One limitation of XML is that it requires all data to be organized in a strict hierarchy. Often, there isn't a single logical hierarchical structuring of the data. For examples, should publications be organized by year, by venue, or by author? Each may be more appropriate for some application scenarios, but XML requires that a single choice be made. TIMBER supports multi-color XML [11], where multiple hierarchies can be established, in different "color", on the same data. This multi-color facility is of particular value in a data warehousing context.

In addition to the macro benchmarks such as XMark [1], in developing an XML database system, we felt the need for a diagnostic benchmark. Traditional application level benchmarks included too many things in a single number so that it was hard for us to determine why performance was bad when we found it to be worse than we expected. We created MBench [20], an engineer's XML benchmark, for ourselves. MBench provides pairs of queries that differ in only one parameter value, thereby providing valuable information regarding what situations hurt performance.

6 Discussion and Conclusion

The heart of TIMBER is its algebra. Having this algebra allowed us to deal with a large subset of XQuery, including nesting, joins, grouping, and ordering, while at the same time enabling optimizations and set based processing. The heterogeneity of XML makes set-oriented processing difficult. The semantics of XQuery are defined in terms of a tuple-at-a-time nested loops structure, and this exacerbates the difficulty. The TIMBER family of algebras provide an elegant bridge across this divide.

Unfortunately, this was not one algebra, but rather a set of algebras. Since the algebra did not come before the query language, the algebra had to be extended to keep pace with language features and optimizations supported. This is as if there were SQL before relational algebra. And then we were to devise a sequence of algebras, RA, RA with grouping and aggregation, RA with cube and ROLAP support, and so on. While this was intellectually the right thing to do, this has kept one early algebra from becoming *THE* standard.

Knowing that the heart of our contribution would be at the algebra level, we consciously chose to focus on the upper layers of the database system, and use a data store for the lower layers. We chose to use Shore [6], because it was such a highly-regarded and widely used academic system. This turned out to be a mistake. For one thing, Shore was an academic project, and the code base was no longer supported by the time we began to use it. For another thing, sizes of main memory, and hence of "interesting" databases had grown substantially between the time Shore was implemented and the time TIMBER was implemented. We kept bumping up against Shore scaling barriers. Finally, a large part of the code in a storage manager such as Shore is devoted to transaction management. This was a feature we ended up never using in TIMBER. So we had a great deal of additional code to carry around without using. After several years, we switched to BerkeleyDB, and that addressed the first two problems above, but the third still remains.

In spite of the challenges mentioned above, Shore was a sufficiently robust engine, and the TIMBER code on top written well enough, that we were able to handle gigabyte size XML documents at a time when commercial native XML companies could only do a few megabytes at best. Since then, there has been significant commercial activity, and we believe many commercial engines, particularly those of relational vendors, will comfortably handle much larger sizes than this.

TIMBER is written in a multiplicity of languages, most importantly in C++ for the query evaluation engine and in C# for the parser and rule-based query optimizer. We were early adopters of Microsoft's Visual Studio

.Net. Its cross-language development facilities worked as advertised for us, with only very minor glitches.

TIMBER code is written in a modular way, and source code is available for free download at [21]. We have had over a 1000 copies of TIMBER downloaded. However, we know anecdotally of at least some who downloaded the source but were unable to build a working executable. We believe we could have had many more users if only we could have constructed a smaller footprint system that was easier to build.

Neither TAX nor XQuery supported updates when we started TIMBER. We did build in some update facilities later, but these continue to feel like a retrofit. The weak support for updates once again highlights that transaction support is unnecessary.

In the document world, people are used to having thousands of documents, each relatively small. When XML is treated as a database, the entire database becomes one document. For the same total size of data, obtained as a product of these two, we could have one very large document or many small documents, or something in between. TIMBER consciously made an effort to support the former, knowing that this was a challenge for other native XML systems with a document processing orientation. This allowed TIMBER to shine, on the one hand, but also made comparisons harder.

In terms of a legacy, the stack-based family of algorithms is the one with the most significant impact among all parts of the TIMBER system. Since its introduction, the stack-based structural join algorithm has inspired a stream of work on structural join algorithms, query optimization techniques, indexing techniques, and result-size estimation techniques for XML. The original paper [2] has been cited 474 times according to Google Scholar to date, and has had dozens of researchers devise improvements.

In conclusion, TIMBER was a large systems project run on a shoe-string. The code is still available and is still being downloaded. It includes many novel ideas, and it certainly taught us a great deal about how to build a database system. However, the TIMBER system itself would have had much greater impact and use if we had found a way to bring it out sooner and smaller.

References

[1] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE Conf.*, Mar. 2002.

[3] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an xml database. In *Proc. SIGMOD Conf.*, 2003.

[4] E. Bertino. An indexing technique for object-oriented databases. In *ICDE*, pages 160–170, 1991.

[5] S. Brenes, Y. Wu, D. Van Gucht, and P. Santa Cruz. Trie indexes for efficient xml query evaluation. In *WebDB*, 2008.

[6] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up Persistent Applications. In *Proc. SIGMOD Conf.*, 1994.

[7] G. H. L. Fletcher, D. Van Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. In *DBPL*, pages 48–65, 2007.

[8] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[9] M. Gyssens, J. Paredaens, D. Van Gucht, and G. H. L. Fletcher. Structural characterizations of the semantics of XPath as navigation tool on a document. In *PODS*, pages 318–327, 2006.

[10] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C.Yu. TIMBER: A native XML database. *VLDB Journal*, 11(4), 2002.

[11] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful xml: one hierarchy isn't enough. In *Proc. SIGMOD Conf.*, 2004.

[12] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. DBPL Conf.*, Sep. 2001.

[13] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[14] C. Kilger and G. Moerkotte. Indexing multiple sets. In *VLDB*, pages 180–191, 1994.

[15] A. Nierman and H. V. Jagadish. Protdb: probabilistic data in xml. In *Proc. VLDB Conf.* 2002.

[16] S. Paparizos, S. Al-Khalifa, A. Chapman, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native system for quering XML. In *Proc. SIGMOD Conf.*, Jun. 2003.

[17] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. *Lecture Notes in Computer Science*, 2490:128–147, 2002.

[18] S. Paparizos and H. V. Jagadish. Pattern tree algebras: sets or sequences? In *Proc. VLDB Conf.*, 2005.

[19] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *Proc. SIGMOD Conf.*, Jun. 2004.

[20] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. The michigan benchmark: towards xml query performance diagnostics. *Inf. Syst.*, 31(2):73–97, 2006.

[21] Timber Group at Univ. of Michigan. Timber system. `http://www.eecs.umich.edu/db/timber`.

[22] W3C DOM Working Group. Document Object Model. `http://www.w3.org/DOM/`.

[23] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *Proc. EDBT Conf.*, Mar. 2002.

[24] Y. Wu, J. M. Patel, and H. V. Jagadish. Using histograms to estimate answer sizes for XML queries. *Information Systems*, 2002.

[25] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proc. ICDE Conf.*, Mar. 2003.

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