A Physical Algebra for XML

Stelios Paparizos[*], Shurug Al-Khalifa, H.V. Jagadish, Andrew Nierman and Yuqing Wu

Abstract

We present a physical algebra for the manipulation of XML in a database. We show how to map logical algebra operators to this physical algebra. We also present several physical algebra identities that are useful for query optimization. This physical algebra is the basis for the implementation of the TIMBER native XML database system at the University of Michigan.

1 Introduction

It is well-accepted that a set-oriented algebra is essential for effective query processing in a database system. We have presented in TAX [5] a tree algebra for bulk manipulation of XML. Our experience with relational databases has taught us that logical algebraic expressions are better implemented not directly, but rather after translation into a physical algebra. For instance, cartesian product is a core operator in the logical relational algebra, while a natural join is not. In the physical relational algebra, natural join is a core operator, with the cartesian product implemented as a special case. Similarly, sorting is an important physical algebra operator, but is not present in the logical algebra at all.

In this paper, we present a physical algebra for XML. This algebra has been developed in the context of the TIMBER [9] native XML database system. We begin by presenting necessary background on TIMBER, and on its logical algebra, TAX, in section 2. We describe the TIMBER physical algebra in section 3, along with an algorithm to map logical algebra operators to physical operators. Specifically, we show that our physical algebra is complete with respect to TAX. Finally, in section 4, we establish several physical algebra identities, and discuss how these could be used for query optimization.

2 Background

2.1 System Architecture

The overall architecture of TIMBER is shown in Figure 1. The Query Parser transforms XQuery [2] input into a TAX logical expression, performing the necessary transformations. The Query Optimizer converts this logical expression into a physical algebraic expression, and uses algebraic identities to consider alternative evaluation plans. It uses cost estimates to choose an evaluation plan for the query and hands this plan off to the Query Evaluator, which implements an access method corresponding to each physical operator.

2.2 Tree Algebra

An XML document is a tree, with each edge in the tree representing element nesting (or containment). To enable efficient processing on large databases, we require set-at-a-time processing of data. In other words, we require a bulk algebra that can manipulate sets of trees: each operator on this algebra would take one or more sets of trees as input and produce a set of trees as output. Using relational algebra as a guide, we can attempt to develop a suite of operators suited to manipulating trees instead of tuples. We have devised such an algebra, called TAX [5].

The biggest challenge in devising this algebra is the heterogeneity allowed by XML and XQuery [2]. Each tuple in a relation has identical structure – given a set of tuples 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

[*]Contact at: S. Paparizos, EECS Dept, Univ. of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122. spapariz@umich.edu

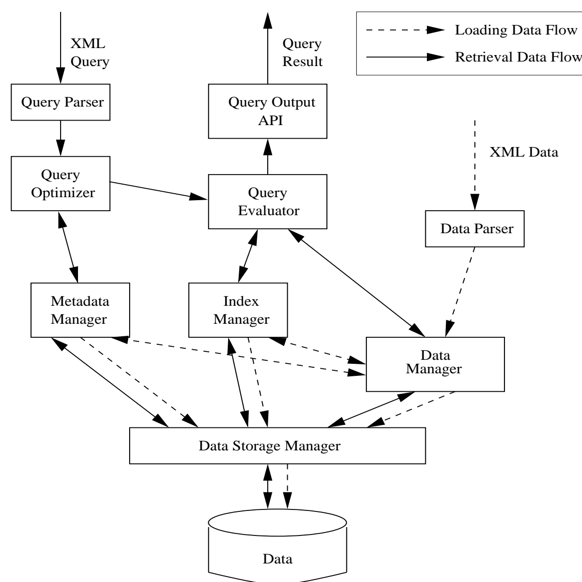


Figure 1: TIMBER Architecture Overview

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.

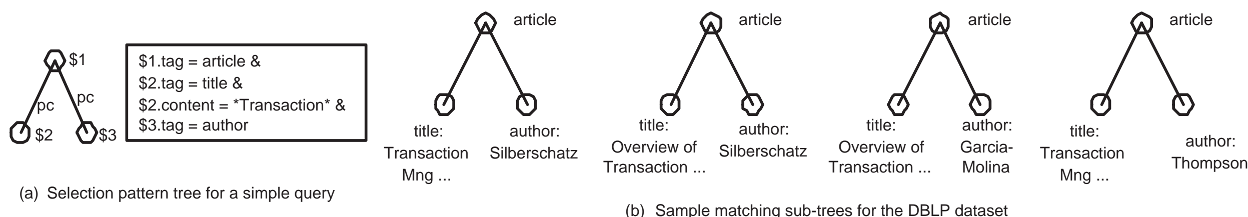


Figure 2: A Pattern Tree for a Query and Witness Trees that Result from a Pattern Match

We solve this problem through the use of *pattern trees* to specify homogeneous tuples of node bindings. For example, a query that looks for articles that have an (at least one) author and a title containing the word "Transaction" is expressed by a pattern tree shown in figure 2.a. Matching the pattern tree to the DBLP database, the result is a set of sub-trees rooted at article, each with author and title. A small sample is shown in figure 2.b. We call such a returned structure a *witness tree*, since it bears witness to the success of the pattern match on the input tree of interest. The set of witness trees produced through the matching of a pattern tree are all homogeneous; we can name nodes in the pattern trees and use these names to refer to the bound nodes in the input data set for each witness tree. A vital property of this technique is that the pattern tree specifies exactly the portion of structure that is of interest in a particular context. All variations of structure irrelevant to the query at hand are rendered immaterial. In short, one can operate on heterogeneous sets of data as if they were completely homogeneous, as long as the places where the elements of the set differ are immaterial to the operation.

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

3 The Physical Algebra

The logical algebra manipulates unordered collections of ordered labeled trees. The physical algebra manipulates ordered collections (sequences) of ordered labeled trees. In consequence, the physical algebra explicitly includes a sort operator to reorder a collection. We discuss here a few noteworthy features of the physical algebra. The complete list of operators can be found in the appendix.

3.1 Pattern Tree Matching

Pattern tree matching is fundamental to the logical algebra. Therefore, every operator includes a pattern tree parameter that is used to identify nodes of interest. This pattern tree matching is not an inexpensive operation and so is the one we would like to minimize the cost of. Fortunately, it turns out that many (but not all) operations in a logical algebraic expression use the same (or small variants of a) common pattern tree. In such a case, we can amortize the evaluation of the pattern tree matching over several physical algebra operators that follow. For this purpose, intermediate results, obtained as the result of a pattern tree match, have nodes annotated with a *node reference* identifying the pattern tree and position in tree that this particular node satisfied. Subsequent physical algebra operators can specify individual nodes in their input tree arguments by means of this node reference, usually as part of a *node reference expression* that specifies the content or a named attribute of the referenced node. In fact, in the physical algebra, it is permissible for the pattern tree match to be accomplished in multiple steps, as described in section 4.

3.2 Order Maintenance

Every physical algebra operator maps one or more sequences of trees to one sequence of trees. In other words, even "set operators", such as union and intersection, operate on sequences. Except where there is an explicit sorting order specified for the output, we retain in the output sequence the order of the input sequence. For instance, binary set operations require that both inputs be ordered in the same manner, and this ordering is preserved in the output. Note that ordering does not necessarily mean sorting based on some attribute value or element content in ascending or descending order: for instance, ordering by increasing value of some hash function is quite appropriate.

A join operation also takes two ordered inputs. It produces output ordered by the left input and for a given value of input further ordered by the right input. A nested-loops join will automatically provide such ordering. A hash-join or sort-merge join would have an ordering operation (such as a sort), applied to each input *prior* to the join itself. Given such input ordering, the output is indeed ordered correctly in the traditional implementations of these access methods.

Most other physical operators take exactly one sequence as input and maintain this order in the output – a natural behavior for any pipelined query processor. An exception are the duplicate elimination and grouping operators. Here, we are explicitly creating output sequences that could be smaller than the corresponding inputs, so order preservation is not natural. While XQuery leaves the output order for duplicate elimination undefined, we prefer to have closure in our algebra and choose to require the output to be ordered by the value(s) used for grouping/duplicate elimination. Once again, recall that ordering by these values does not have to mean sorting by them. A little thought will convince the reader that our requirement of ordering by value is very weak indeed for duplicate elimination, where there can be at most one element in the result set with this value, and for grouping, where all elements with this value are grouped into a single structure for the output result set.

3.3 Procedural Construction

The logical algebra has rename, reorder, and copy-and-paste operators, which, along with projection, are used to obtain required tree manipulations in a declarative fashion. At the implementation level, once an individual tree has been identified for manipulation, there is no benefit in having a declarative set-oriented specification of this manipulation. One may as well have a procedural statement of the manipulation required, and implement this directly. To this end, we introduce a procedural construction operator in the physical algebra. The central restriction on this operator is that it is one-to-one. Given an sequence of input trees, it manipulates each tree in the input *independently* according to a procedural specification to produce exactly one output tree in order. Not coincidentally, this physical operator conveniently maps much of[1] the arbitrary output creation possible in the XQuery CONSTRUCT operator.

[1] In XQuery, construct might include a nested FLWR statement; this will be mapped to a combination of physical algebra operators

3.4 Logical to Physical Mapping

Definition 3.1 A: physical algebra expression $f(p_1, p_2, ...)$ is said to be *equivalent* to a logical algebra expression $e(r_1, r_2, ...)$, if for every sequence p_i there corresponds an unordered collection r_j (and vice versa), such that whenever each p_i comprises exactly the same element trees as the corresponding r_j, the result of evaluating $f(p_1, p_2, ...)$ is the same, ignoring order, as that of $e(r_1, r_2, ...)$. $\diamondsuit$

Theorem 3.1 *For every TAX logical expression e, there exists an equivalent physical algebra expression f.* $\diamondsuit$

The proof is by exhaustive enumeration of the TAX operators. A couple of these are shown below to give the reader an idea of what is involved.

3.4.1 Selection

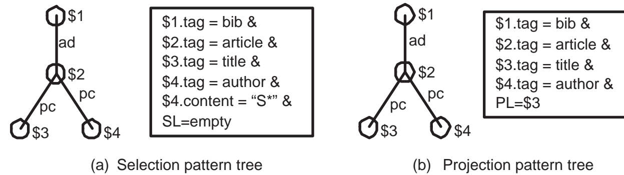
Let $\delta_{p,f}(S, i)$ be the data access operator. It takes as input a sequence of trees S, an index name i, a predicate p and a sub-tree flag f. It outputs all nodes in the tree that satisfy p and, if f is true, the subtree rooted at the corresponding node. Let $\epsilon_{b,(r_1,...,r_{n-1}),nl}(S_0, S_1, \ldots, S_{n-1})$ be the structural join operator. It takes as input n sequences of trees S_i, $n-1$ structural relationships with node references r_i, a sorting basis b and a list of node reference expressions nl to be used by the internal projection. The operator joins each tree in S_i and S_j, with $i < j$, by using information from the structural relationship r_j. The output is sorted based on b and only the nodes specified in nl are retained. When $b = any$, the internal sorting procedure is not that important and any basis can be used to sort the output. When $nl = \emptyset$, the entire tree is returned. Using these we write:

$$Select_{P,SL}(D) = \epsilon_{any,(r_1,...,r_n),\emptyset}\{\delta_{p_0,sl_0}(D, i), \ldots, \delta_{p_n,sl_n}(D, i)\} \tag{1}$$

The select logical operator takes as input a pattern tree P and a selection list SL. In our mapping the δ operator will retrieve each node in the pattern tree. Within each δ operator, p_i corresponds to the i node in the pattern tree and sl_i is true (T) only if the node is included in the selection list SL. Then all those individual nodes are joined using the ϵ operator. The output order is not important for the logical operator so the output is sorted by any; also all the output nodes will be projected out. Each structural relationship r_j corresponds to a relationship[2] between node p_i and node p_j, with $i < j$.

To better understand how the mapping works, let's consider the following example. We want to apply a selection on the database using pattern tree P_A, as shown in figure 3.a. This mapping will be:

$$\begin{aligned} Select_{P_A,SL_A}(D) \quad = \quad & \epsilon_{any,[(ad,bib,book),(pc,book,title),(pc,book,author)],\emptyset}\{\delta_{bib,F}(D, i), \\ & \delta_{book,F}(D, i), \delta_{title,F}(D, i), \delta_{author=\text{“}S*''\!,F}(D, i)\} \end{aligned} \tag{2}$$



(a) Selection pattern tree (b) Projection pattern tree

Figure 3: Sample selection and projection pattern trees used by the selection and projection logical operators

3.4.2 Projection

Let $\pi_{nl}(S, T)$ be the project physical operator. It takes as input a sequence of trees S, a selection procedure T and a list of node reference expressions nl. The selection procedure T is used to describe the nodes of each tree in S that will be bound with references. If $T = \emptyset$ then the binding has already occurred and all the node reference expressions in nl can be mapped correctly. The operator retains the nodes in each tree of S as specified by nl and projects out the remaining ones. Using this operator we can write:

$$Project_{P,PL}(D) \quad = \quad \pi_{PL}\{D,\ \epsilon_{any,(r_1,...,r_n),\emptyset}[\delta_{p_0,F}(D, i), \ldots, \delta_{p_n,F}(D, i)]\} \tag{3}$$

[2] ancestor-descendant (ad) or parent-child (pc)

The project logical operator takes as input a pattern tree P and a projection list PL. The nodes that correspond to the pattern tree P have to be bound to node references. This is necessary for the projection operator π to be able to use the list of node reference expressions $nl = PL$ to retain the appropriate nodes. Nodes in the pattern tree are referred as p_i. Relationships r_j refer to the structural relationship[2] between nodes p_i and p_j, with $i < j$.

To better illustrate how projection works let's apply a project operation using the projection tree P_B and PL_B as shown in figure 3.b. The physical mapping of this project operator would be:

$$\begin{aligned} Project_{P_B,title}(D) &= \pi_{title}\{D,\ \epsilon_{any,[(ad,bib,book),(pc,book,title),(pc,book,author)],\emptyset}[\\ &\quad \delta_{bib,F}(D,i), \delta_{book,F}(D,i), \delta_{title,F}(D,i), \delta_{author,F}(D,i)]\} \end{aligned} \tag{4}$$

3.5 Non-minimality

The set of physical algebra operators is not minimal. For instance, duplicate elimination is a special case of grouping, and projection is a special case of procedural construction. We find no strong reason to keep the set of physical algebra operators minimal when some special cases, or composite operators, can be implemented more efficiently. Explicit identification of such possibilities through algebraic rewriting is a central purpose of having a physical algebra.

4 Algebraic Identities

The primary reason to develop a physical algebra is to have a means to optimize evaluation of complex queries through algebraic rewriting. We list a few interesting algebraic identities below:

$$\pi_{pl}\{\epsilon_{any,(r_1,\ldots,r_n),\emptyset}(S_0,\ldots,S_n),\emptyset\} = \epsilon_{any,(r_1,\ldots,r_n),pl}(S_0,\ldots,S_n) \tag{5}$$

$$m \leq n, \quad \pi_{pl}\{\epsilon_{any,(r_1,\ldots,r_n),\emptyset}(S_0,\ldots,S_n), \epsilon_{any,(r_1,\ldots,r_m),\emptyset}(S_0,\ldots,S_m)\} = \pi_{pl}\{\epsilon_{any,(r_1,\ldots,r_n),\emptyset}(S_0,\ldots,S_n),\emptyset\} \tag{6}$$

$$\epsilon_{any,(r_1,\ldots,r_n),\emptyset}(S_0,\ldots,S_n) = \epsilon_{any,(r_q),\emptyset}\{[\epsilon_{q-1,(r_1,\ldots,r_p),\emptyset}(S_0,\ldots,S_{q-1})], [\epsilon_{q,(r_{q+1},\ldots,r_n),\emptyset}(S_q,\ldots,S_n)]\} \tag{7}$$

$$\tau_{A,asc}\{\epsilon_{B,(r_1),\emptyset}(S_A,S_B)\} = \epsilon_{A,(r_1),\emptyset}(S_A,S_B) \tag{8}$$

$$\cup(S_A,S_B) = \cup(S_B,S_A) \tag{9}$$

$$\cap(S_A,S_B) = \cap(S_B,S_A) \tag{10}$$

$$\cup[\epsilon_{A,(r_{AB}),\emptyset}(S_A,S_B), \epsilon_{A,(r_{AC}),\emptyset}(S_A,S_C)] = \epsilon_{A,(r_{A\cup}),\emptyset}(S_A,\cup(S_B,S_C)) \tag{11}$$

$$\backslash[\epsilon_{A,(r_{AB}),\emptyset}(S_A,S_B), \epsilon_{A,(r_{AC}),\emptyset}(S_A,S_C)] = \neg\epsilon[\epsilon_{A,(r_{AB}),\emptyset}(S_A,S_B), r_{AC}, S_C] \tag{12}$$

$$m \leq n, \quad \epsilon_{any,r,\emptyset}\{\delta_{p_1,F}[\epsilon_{any,r,\emptyset}(\delta_{p_1,F}(D,i),\ldots,\delta_{p_n,F}(D,i)), i],\ldots,\delta_{p_m,F}[\epsilon_{any,r,\emptyset}(\delta_{p_1,F}(D,i),\ldots,\delta_{p_n,F}(D,i)),i]\} = \epsilon_{any,r,\emptyset}[\delta_{p_1,F}(D,i),\ldots,\delta_{p_m,F}(D,i)] \tag{13}$$

Equation 5 describes the procedure of pushing projections in a structural join. Equation 6 demonstrates the case when the requested binding procedure does not need to apply, because the necessary nodes have already been bound. Equation 7 describes how any structural join with $n > 3$ input sequences can be broken into smaller joins. Equation 8 describes how an external sort can be pushed in the structural join. Equations 9, 10 demonstrate that union and intersection are symmetrical operators. Equations 11, 12 demonstrate how to push union and difference in a structural join, when the two set operators are applied on input sequences whose trees AB, AC have a common part A. Equation 13 shows that δ operators to get a specific node do not need to be applied more than once in the same execution involving structural joins.

To better understand how these identities optimize the evaluation of complex queries through algebraic rewriting, we will consider the example of a project operator applied after a select operator using the same pattern tree, as seen in figure 3. What is a very complicated execution is transformed into a simple one with obvious benefits in the complexity and the performance of the execution.

$$\begin{aligned} Project_{P,title}(Select_{P,\emptyset}(D)) &= \pi_{title}\{\epsilon_{any,[(ad,bib,book),(pc,book,title),(pc,book,author)],\emptyset}[\delta_{bib,F}(D,i), \\ &\quad \delta_{book,F}(D,i), \delta_{title,F}(D,i), \delta_{author=``S*'',F}(D,i)], \end{aligned}$$

$$
\begin{aligned}
& \epsilon_{any,[(ad,bib,book),(pc,book,title),(pc,book,author)],\emptyset}[\\
& \delta_{bib,F}(D,i), \delta_{book,F}(D,i), \delta_{title,F}(D,i), \delta_{author,F}(D,i)]\} \\
applying\ (6) \quad = \quad & \pi_{title}\{\epsilon_{any,[(ad,bib,book),(pc,book,title),(pc,book,author)],\emptyset}[\delta_{bib,F}(D,i), \\
& \delta_{book,F}(D,i), \delta_{title,F}(D,i), \delta_{author=``S*'',F}(D,i)], \emptyset\} \\
applying\ (5) \quad = \quad & \epsilon_{any,[(ad,bib,book),(pc,book,title),(pc,book,author)],title}[\delta_{bib,F}(D,i), \\
& \delta_{book,F}(D,i), \delta_{title,F}(D,i), \delta_{author=``S*'',F}(D,i)] \qquad (14)
\end{aligned}
$$

After the algebraic rewrite, the query optimizer will perform a cost based analysis to select the most efficient query execution plan. To do the analysis the optimizer will generate variations of (14) using identity (7) appropriately. So the plan with the best execution performance will look like:

$$
\begin{aligned}
& \epsilon_{any,(ad,bib,title),title}\{\delta_{bib,F}(D,i), \epsilon_{any,(pc,book,title),title}[\delta_{title,F}(D,i), \\
& \quad \epsilon_{any,(pc,book,author),book}(\delta_{book,F}(D,i), \delta_{author=``S*'',F}(D,i))]\} \qquad (15)
\end{aligned}
$$

5 Related Work and Conclusion

There is no shortage of algebras for data manipulation; even in the XML context, several algebras have been proposed. The W3C working group on XML Query has recently issued an algebra document [1]. The focus of this algebra is to provide a formal semantics for XQuery [2]. It is not suitable for set-oriented processing.

There are a few examples of tree targeted algebras. The Aqua project [3], which focuses on the identification of pattern matches and their rewriting in the style of grammar production rules. A grammar-based algebra [4], shown equivalent to a calculus, for manipulating tree-structured data using production rules. An extension of a relational algebra [6] that uses binding of sets of tuples with specified labeled nodes. Similarly, a navigational algebra [7], treating individual nodes as the unit of manipulation. And the algebra [8], used as the basis for the Niagara XML data management system. However none of the preceding suggestions makes the distribution between logical and physical algebra.

Experience with relational databases has taught us that logical algebraic expressions are better implemented not directly, but rather after translation into a physical algebra. We have presented a physical algebra for the manipulation of XML in a database. We have shown how to map logical algebra operators to this physical algebra and presented several physical algebra identities that are useful for query optimization.

References

[1] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon and P. Wadler. XQuery 1.0 Formal Semantics. W3C Working Draft, June 7, 2001. Available at `http://www.w3.org/TR/query-semantics/`

[2] D. Chamberlin, D. Florescu, J. Robie, J. Simeon and M. Stefanescu. XQuery: A Query Language for XML. W3C Working Draft, Dec. 2001. Available at `http://www.w3.org/TR/xquery`

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

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

[5] 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.

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

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

[8] L. Galanis, E. Viglas, D.J. DeWitt, J.F. Naughton and D. Maier. Following the Paths of XML Data: An Algebraic Framework for XML Query Evaluation. 2001. Available at `http://www.cs.wisc.edu/niagara/papers/algebra.pdf`

[9] University of Michigan, The TIMBER system. `http://www.eecs.umich.edu/db/timber/`

A List of Physical Algebra Operators

DataAccess $(\delta_{p,f}(S,i))$:

input A sequence of trees S, predicate p, (optional) index name i, sub-tree flag f

output A sequence of trees

order Preserved

description For each input tree, it outputs each node that satisfies a specified predicate. Can be performed through an indexed access or a scan is no index name is provided. The input can be a single tree database. When the subtree flag is set on, each output element in the subtree rooted at the corresponding node will be included in the output. The operator binds a node reference with the root node of each tree in the output tree sequence.

Filter $(\phi_p(S))$:

input A sequence of trees S, predicate p

output A sequence of trees

order Preserved

description Given a sequence of trees, output only the trees in this sequence that satisfy the specified filter predicate.

Sort $(\tau_{b,m}(S))$:

input A sequence of trees S, Sorting basis vector b, Mode m

output A sequence of trees

order Based on the internal sorting procedure

description Sorts the input sequence based on the sorting basis vector and the requested mode (Ascending/Descending). Each element of the vector is a Node Reference Expression (NRE)[3]. The output order sequence to reflects the sorting procedure that took place.

ValueJoin $(\chi_p(S_l,S_r))$:

input A sequence of trees S_l, A sequence of trees S_r, Join predicate p

output A sequence of trees

order Based on the order of the S_l input sequence

description Performs a value-based join on the two input sequences specified by the join predicate. The procedure can be described by a nested for loop where each S_l (outer) tree is matched with the corresponding S_r (inner) tree(s) using a value-based comparison as specified by the join predicate. The output sequence order is based on the left input sequence order.

StructuralJoin $(\epsilon_{b,(r_1,\ldots,r_{n-1}),nl}(S_0,S_1,\ldots,S_{n-1}))$:

input N sequences of trees $S_1 \ldots S_n$, (N-1) Structural relationships with node references $r_1 \ldots r_{n-1}$, Sorting basis b, A list of node reference expressions nl to be used by the internal projection

output A sequence of trees

order Based on the order of the internal sorting

description The input number of sequences must be $N \geq 2$. Any relation r_i relates a node of S_i with a node of S_j such that $i > j$. The input sequences must be sorted based on the NRE of the desired structural relationship. Joins n input trees based on the structural relationships between them (ancestor-descendant or parent-child for each pair). The output may include any m nodes of the n inputs, and may be sorted by any of the m outputs. If nl is $\emptyset$ then all nodes are included in the output. The output sequence order reflects the sorting that took place. The operator binds a node reference with the nodes that participate in the structural joins.

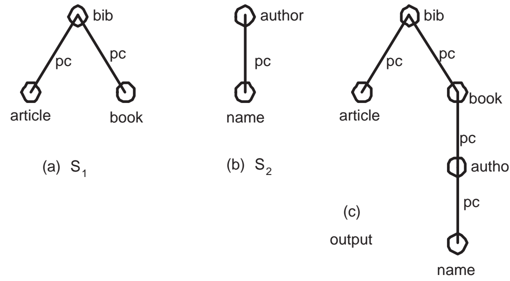


Figure 4: An example for structural join: $\epsilon_{any,(pc,book,author),\emptyset}(S_1, S_2)$

example An example is shown in figure 4.

BinaryNegativeStructuralJoin $(\neg\epsilon(S_l, r, S_r))$:

input A sequence of trees S_l, A sequence of trees S_r, Structural relationship with node references r

output A sequence of trees

order Based on the order of the internal sorting

description Joins two inputs based on one not being the descendant/child of the other. Each output tree produced contains only the tree with the ancestor/parent. The output is sorted by the node id of the ancestor/parent[4]. The operator binds a node reference with the nodes that participate in the structural join.

Project $(\pi_{nl}(S, T))$:

input A sequence of trees S, A selection procedure T, A list of node reference expressions nl

output A sequence of trees

order Preserved

description The selection procedure T is used to describe the nodes of each tree in S that will be bound with references. If $T = \emptyset$ then the binding has already occurred and all the node reference expressions in nl can be mapped correctly. The operator retains the nodes in each tree of S as specified by nl and projects out the remaining ones.

DuplicateEliminator $(\xi_n(S))$:

input A sequence of trees S, Node reference expression n

output A sequence of trees

order Based on sorting by the given NRE

description Eliminates duplicates on the given sequence of trees based on the content of the referenced nodes. To eliminate the duplicates the input sequence is first sorted by the given NRE. So the output order reflects this internal sorting.

Union $(\cup(S_l, S_r))$:

input A sequence of trees S_l, A sequence of trees S_r

output A sequence of trees

order Based on the merging procedure of the two input sequences

description The input sequences must be sorted on the same basis. Merges the trees in the two input sequences producing a union output sequence.

[3] An NRE can be the content (text) of an element or an attribute, the tag name of an element or the node id

[4] The parent can be on both sequences

Intersection $(\cap(S_l, S_r))$:

input A sequence of trees S_l, A sequence of trees S_r

output A sequence of trees

order Based on either input sequence.

description The input sequences must be sorted on the same basis. Iterates over the two input sequences and outputs their common trees. The output order sequence is depended on either input sequence.

Difference $(\backslash(S_l, S_r))$:

input A sequence of trees S_l, A sequence of trees S_r

output A sequence of trees

order Based on the S_l input sequence.

description The input sequences must be sorted on the same basis. Computes the set difference between from the two input sequences $(tree_i S_l - tree_j S_r)$.

GroupBy $(\gamma_{gb,sb}(S))$:

input A sequence of trees S, Grouping basis vector gb, Sorting basis vector sb

output A sequence of trees

order Based on the grouping basis gb, internally each tree is sorted by the sorting basis sb.

description The grouping and sorting basis vectors are using node reference expressions. Sorts the input sequence using the grouping NRE. Groups the trees together based on the grouping NRE. Creates dummy nodes to be used as the grouping root, the grouping sub-root and the grouping basis. Creates output trees containing the dummy nodes, the grouping basis and the corresponding grouped trees. Internally in each output tree the grouped trees are sorted using the sorting basis. The output tree sequence order is based on a sort by the grouping basis. The operator binds a node reference with the nodes that participate in the grouping.

example An example is shown in figure 5.

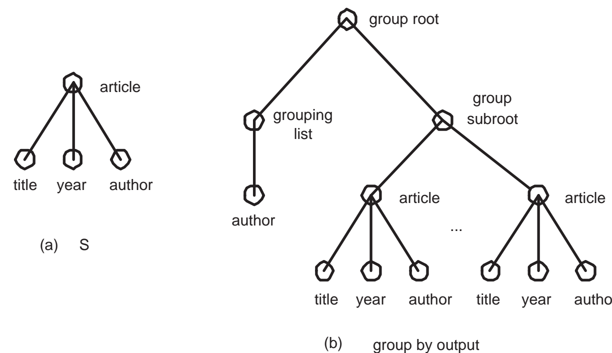


Figure 5: An example for group by: $\gamma_{author,any}(S)$

AggregateFunction $(\xi_{in,on}(S, name))$:

input A sequence of trees S, Input node reference expression list in, Output node reference expression on, Aggregate function name $name$

output A sequence of trees

order Preserved

description It processes one tree at a time and the aggregation takes place within each tree. The input NREs point to the sub-tree nodes used for the aggregation. The output NRE points to the place where the output value will be stored. Probably a grouping will take place before an aggregation function can be applied. A construct will create a dummy node for the result to be placed.

Construct ($\kappa_c(S)$) :

input A sequence of trees S, Construction operation description c

output A sequence of trees

order Preserved

description Apply the construction operation c on each tree of S. Can create dummy nodes and add them to each tree. Can reorder nodes within each tree. Can rename nodes within each tree. Can project specific nodes of each tree. The operator binds a node reference with the nodes that are added in each tree.