Pattern tree algebras: sets or sequences?

Stelios Paparizos*  
H. V. Jagadish*

University of Michigan  
Ann Arbor, MI, USA  
{spapariz, jag}@umich.edu

Abstract

XML and XQuery semantics are very sensitive to the order of the produced output. Although pattern-tree based algebraic approaches are becoming more and more popular for evaluating XML, there is no universally accepted technique which can guarantee both a correct output order and a choice of efficient alternative plans.

We address the problem using hybrid collections of trees that can be either sets or sequences or something in between. Each such collection is coupled with an Ordering Specification that describes how the trees are sorted (full, partial or no order). This provides us with a formal basis for developing a query plan having parts that maintain no order and parts with partial or full order.

It turns out that duplicate elimination introduces some of the same issues as order maintenance: it is expensive and a single collection type does not always provide all the flexibility required to optimize this properly. To solve this problem we associate with each hybrid collection a Duplicate Specification that describes the presence or absence of duplicate elements in it. We show how to extend an existing bulk tree algebra, TLC [12], to use Ordering and Duplicate specifications and produce correctly ordered results. We also suggest some optimizations enabled by the flexibility of our approach, and experimentally demonstrate the performance increase due to them.

1 Introduction

XML means many things to many people, and gets used in a variety of ways. The formal semantics of XML and XQuery require ordering, yet 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 fact, we would like even more. Even when ordered final results are required, it may not be necessary to maintain order at each intermediate step. Exploiting this flexibility, provided the required order can eventually be established (or recovered), can lead to significant performance benefits.

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. It is not difficult to show that just manipulating sets is not enough; we often do need to consider order in intermediate results. Yet we know that manipulating sequences is considerably more expensive, so we would prefer not to work with these if we can avoid it.

The solution we propose in this paper is to define a new generic collection type, which could be a set or a sequence or even something else. We associate with each collection an Ordering Specification that indicates precisely what type of order, if any, is to be maintained in this collection. We then develop an algebra for manipulating collections with ordering specifications. Using this algebra we are able to develop query plans that maintain as little order as possible during query execution while yet producing the correct query results.

Duplicates in collections are also a topic of interest, not just for XML, but for relational data as well. In relational query processing, duplicate removal is generally considered expensive, and avoided where possible even though relational algebra formally manipulates sets that do not admit duplicates. In fact, there has been considerable work towards developing multi-set relational algebras. 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. It turns out that our proposal of using a generic collection type applies to this problem as well, through associating an explicit Duplicate Specification with each collection. We are then able to develop an algebra for manipulating such collections, and use this algebra to develop query plans in which duplicate elimination is optimized.

*Supported in part by NSF under grants IIS-0208852 and IIS-0219513.
In Section 6 we use T features on both duplicates and output order in Section 5.

The rest of the paper is organized in the following way: our discussion on ordering and duplicates in the paper starts in Section 2 by identifying the requirements set by XML and XQuery. Then we introduce our proposed solution to the problem that uses hybrid collections paired with the Ordering Specification O-Spec and Duplicate Specification D-Spec in Section 3. We then consider a set of common operators in a bulk tree-algebra (such as TLC [12]), study their behavior in relation to duplicates and order of output in Section 4.2 and show how such an approach can generate the correct results in Section 4.4. We continue by showing potential optimizations that take advantage of our features on both duplicates and output order in Section 5. In Section 6, we use TIMBER [10], a native XML system, to experimentally illustrate the benefits of using our solution and the gained performance increase.

2 Ordering & Duplicates in XML / XQuery

Before we discuss XML and XQuery ordering requirements, we start with a few observations on XQuery set semantics (presence of duplicates). XQuery associates variables via the FOR and LET clauses. The LET clause creates a variable binding with an entire set of matching XML elements (nodes) whereas the FOR clause creates a binding with each element (node) of a set of matching XML elements (nodes). We can see that both binding types require a duplicate-free set of matching XML elements to be calculated first. Hence XQuery requires all duplicates to be removed when creating the variable assignments to the corresponding XML parts.

XML itself incorporates semantics in the order data is specified. XML queries have to respect that and produce results based on the order of the original document (queries based on XQuery or XPath adhere to that). 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. To better understand the issues with binding order an example is shown in Fig. 1. Two very simple queries are shown – both perform a join between book and article elements on author. The output of the first is sorted on the original document order of {book, article} whereas the output of the second is sorted on the original document order of {article, book}. Notice how article and book switched places to reflect the difference in the order requirements of the two queries, although the rest of the query remained the same.

In more complex scenarios with implicit ordering (including nested queries) binding order is much harder to follow correctly and efficiently.

A FLWOR statement in XQuery may include an explicit ORDERBY clause, specifying the ordering of the output based on the value of some expression – this is similar in concept with ordering in the relational world and SQL. Note that the ORDERBY clause, when present, overrides document and binding order for that single-block FLWOR statement.

To facilitate our discussion in the rest of the paper, we present a summary of XQuery ordering requirements in Fig. 2. An XML processing system must be able to understand and process these requirements properly. Existing XML algebras tried to produce the correct output order by using sets or sequences. Sets lose all ordering information resulting in redundant sorts throughout the query plan, whereas sequences maintain the order throughout the query plan but make it difficult to perform rewrites and need tightly bound operators that can be expensive to implement and difficult to optimize. Also, whether sets or sequences are used, semantics require no duplicates are present and so require several duplicate elimination operations to be applied through the plan. These problems motivate our current work.

3 Introducing the Hybrid Collection

In this section we introduce the core of our solution to the problem, the concept of a generic collection type with ordering and duplicate specifications. We believe that the basic principles of our collections can be incorporated into any bulk algebra, with the appropriate modifications. For our presentation we choose to follow the tree-based algebraic approach and model our collections to operate with tree structures. A tree consists of nodes, with each node mapping to an XML element or attribute. A basic assumption we make is that the nodes used are annotated with identifiers that guarantee both uniqueness and document order.

Definition 3.1 Given a collection of trees $C_T$, the Duplicate Specification D-Spec describes how trees were determined as identical and were eliminated from the collection $C_T$. The value of D-Spec can be one of the following:

- empty $\{\}$: Any type of duplicate tree can potentially be present in the collection $C_T$.
- full tree $\{V_i, E_i\}$: Given a tree $Q_i = (V_i, E_i)$ in the collection $C_T$, there does not exist another tree $Q_j = (V_j, E_j)$ in the collection $C_T$ such that $V_i = V_j$ and $E_i = E_j$. $(\forall Q_i, \exists Q_j (Q_i, i \neq j) : |[V_i = V_j) \land (E_i = E_j)]|$
Example 3.1: A group of sample collections with D-Spec describing how duplicates were previously removed from them can be found in Figure 3.

Notice the duplicates that exist in part (1), how the last tree from (1) is removed in part (2) and how multiple trees are removed in part (3) – potentially losing some information about A (had the trees with only A removed we would have lost all A2 from the collection).

As we can see, the partial-tree comparison can remove more trees than deep-tree. This option is designed to be used for optimization purposes. We can produce the correct output without it, but this would lead to a less flexible solution with redundant duplicate elimination operations with various ‘tree’ inputs.

The goal of Duplicate Specification D-Spec is to carry to succeeding operators the information of a duplicate elimination procedure that was applied earlier in a query plan, no matter whether such procedure was explicit or implicit. Given the D-Spec for each collection the optimizer can apply it on the plan either globally or locally. Globally by rewriting the plan and choosing where to place duplicate elimination procedures to achieve the correct result with a minimum cost. Locally by considering each operator and choosing during the physical evaluation to: a) ignore the D-Spec for operators that do not remove or add duplicates, b) modify the D-Spec appropriately for operators that remove or produce duplicates, and c) maintain the D-Spec for operators that require duplicate elimination but use D-Spec to determine that it was already applied.

Definition 3.2: An Ordering Item O-Item is the unit used when sorting a collection of trees. It consists of three parameters: i) a reference to identify the node in the tree to sort the trees by, ii) ascending (‘asc’) or descending (‘desc’) describes how to use the node information to perform the sort and iii) empty greatest (‘g’) or empty least (‘l’) describes where to place the trees without a matching node for the given input reference.

O-Item uses a flag to refer to the identifier or content of each node. As we assumed the node identifiers used in conjunction with our collections can indicate document order for any group of nodes, we can use the node identifiers to indicate ordering based on document order. When the required order is based on a value comparison the node content can be accessed instead.

An Ordering Item is essentially the smallest unit that can be used to describe a sorting procedure. It matches in principle the input to a SORT operation. It contains the means to identify the sorting basis (what to sort on), how to use the information in the basis to perform the sort (e.g. ascending and what to do with null entries (place them in the beginning or end of the collection). Complex sorting procedures that require multiple parameters to perform the sort can easily be described using a list of Ordering Items.

Example 3.2: For example a simple sorting procedure using all B nodes (identifiers), sorting them on ascending order and placing all empty entries in the beginning will be described by an O-Item that looks like (B, asc, l). For simplicity, ascending (‘asc’) and empty least (‘l’) are the default choices and can be omitted; so (B, asc, l)→(B).

Definition 3.3: Given a collection of trees CT, the Ordering Specification O-Spec specifies how the trees are sorted.
in the collection $C_T$. O-Spec accepts as input a list of Ordering Items O-Items describing the sorting procedure that occurred on the collection. Sorting based on each O-Item happens in the order they are given in the input list.

Similar in spirit with the Duplicate Specification, the Ordering Specification is used in association with a collection to describe a sorting procedure (implicit or explicit) that took place earlier in the query plan. The optimizer can use the information globally by rewriting the plan and choosing where to place sorts to achieve the correct result while using a ‘quick’ query plan. It can also use them locally during the physical evaluation and determine whether: a) an operator can pass it along if it does not change the order of the collection, b) modify it with a partial new sort, and c) create a completely new one when it destroys the order of the collection and/or reorders it. 

**Example 3.3** In Fig. 4, we can see a few ordered collections using the Ordering Specification. In part (1) we see a “fully-ordered” collection; all the nodes in every tree were used to perform the sort. A “fully-ordered” collection has one and only one way that the trees can be ordered (absolute order). In parts (2.a) and (2.b) we see the same “partially-ordered” collection; only nodes in parts of every tree were used to perform the sort. A “partially-ordered” collection can potentially have multiple ways it was ordered. Parts (2.a) and (2.b) show the same collection ordered by the same key with clearly more than one representations of the absolute tree order. In part (3) we see a collection with unspecified order (any order).

Ordering Specification is orthogonal to the presence of duplicates and Duplicate Specification. To clarify whether a collection is an arbitrary one or it is coupled with an Ordering or Duplicate Specification we accordingly use the terms D-collection, O-collection and OD-collection.

An outcome of our technique is that the Ordering Specification O-Spec of a collection (and for that matter the SORT operation that produced it) is a superset of the potential order that can be expressed by XQuery. The output of a single block FLWOR statement in XQuery can be ordered by either the binding/document order as specified in the FOR clauses or the value order as specified in the ORDERBY clause. O-Spec allows for the result of a query to be sorted by the combination of binding/document and value order. In other words, despite the XQuery complex order requirements, there is no way to write a simple query that asks to “return all books, sorted by the document order of each author and the value order of each year”. Our approach can easily handle this by specifying to access the node id (provides document order) for author and the node content (provides value order) for year in their corresponding Ordering Items as part of a SORT operation. In that sense, our approach can provide the means for more powerful ordering expressions for XML than XQuery.

**4 Enhancing an Algebra**

Hybrid collections can be used to enhance any algebra. We focus on the TLC [13] algebra used in the TIMBER [1] system. In this section we describe how TLC algebraic operators can be extended to handle order. We show some algebraic identities that can be generated and present the algorithm to produce the correct output order. But first we begin with some necessary background on TLC.

**4.1 Tree Logical Classes (TLC) for XML**

All tree algebras (e.g. [3] [8]) use pattern tree match as a basic building block. A pattern tree is matched on an XML document and a collection of witness trees is produced. In previous tree algebra solutions, the witness trees had 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. TLC uses Annotated Pattern Trees (APTs) and Logical Classes (LCs) to overcome that limitation. It is not within the context of this paper to present in detail how TLC [13] works. Instead we will try to show the intuition on APTs and LCs and walk through an example with them.

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). Fig.5 shows the example match for an annotated pattern tree. The figure illustrates how annotated pattern trees address heterogeneity on both dimensions (height and width) using variations of annotated edges. So $A_1$, $A_2$ and $E_3$ are matched into clustered siblings due to the “*” and “*” edges in the APT. On the flip side $D_1$, $D_2$ matchings will produce two witness trees for the first input tree (the second tree is let through, although there is no D matching) due to the “?” edge in the APT.

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 Fig.5 how can we say precisely which node we mean? The solution to his 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 Fig.5, the red(gray) circle indicates how the A nodes form a logical class for each.
Given an input hybrid collection of trees, the predicate with similar properties for uniqueness and document order RETURN a constructed node (e.g. a node created for a tag tagging, renaming, and arbitrary tree assembly. Any annotated pattern tree (APT), except it allows facilities for input: output one tree for each tree in the tree annotated by newLC.

Join J[apt, p, ord](C_t, C_r): Given two input hybrid collections of trees C_t and C_r, an Annotated Pattern Tree apt, a predicate p and an ordering parameter ord: output one tree for each tree from C_t that has a matching tree from C_r as described in p. The structure of the output tree matches that of the apt. The output is ordered as specified by ord: a) ‘empty’, the output order is unspecified and output O-Spec is empty. b) ‘maintain’, maintains the order of the input (left, right), the output O-Spec is the combination of the left and right input. c) ‘list-resort’, output is resorted based on the list of nodes passed – the input order is destroyed. d) ‘list-add’, output is partially sorted based on the list of nodes passed – the input order (left, right) is maintained and the additional parameters from ord are added to the combination O-Spec of the inputs.

Reorder R[pLC, cLC, f](C_t): accepts an input collection C_t, two Logical Classes pLC and cLC and a function f. For each tree in C_t, identify the parent nodes pLC and the child nodes cLC then reorder cLC under each pLC as specified by the input function f.

Duplicate-Elimination DE[dep](C_t): Given an input hybrid collection of trees C_t and a parameter dep: output a collection from C_t having removed identical trees compared based on dep. dep can be a) the whole tree (input ‘tree’) or b) a list of nodes (input dl). The output D-Spec becomes dep.

Sort O[ol](C_t): Given an input hybrid collection of trees C_t a sorting basis vector ol: output collection C_t reordered as described by ol. Each entry for ol is an Ordering Item O-Item. The output O-Spec becomes ol. Sort can choose to partially order the input collection if the input O-Spec is a subset of ol.

4.3 Algebraic Identities

The power of bulk algebra is in the rewrites that rely on algebraic identities. In Table 3 we present several identities showing how our operators interact with Sort and Duplicate Elimination. We try to give the intuition on how these identities are generated.

Select and Join accept an optional ordering parameter that allows us to push Sort into them. Also, they can be set to maintain order and be swapped with Sort. Identities (1), (2), (3), (4) are generated because of this.

Project removes nodes from a tree and thus can create duplicates. The removed nodes might affect succeeding ordering operations. So both ordering and duplicates must be aware of the project list, hence identities (5),(6). Construct is somewhat similar to that because it can also remove some nodes. So identity (7) is generated.

Filter does not modify the tree structure at all. So it
is dealt with. The query asks for all simplified on purpose to allow focus on how binding order

Example 4.1 using an example.

We will try to discuss the intuition behind the algorithm

Pseudocode for the extension algorithm is shown in Fig. 6.

A bulk algebra can be extended to produce the correct or-

does not add or remove nodes either, but modifies the tree
structure by reordering it. Yet, all the tree information is
retained and the modification is applied the same way to
each tree; trees that were identical before reorder, become
identical again. So (10) and (11) can be used.

Construct can eliminate duplicates due to the con-
structed nodes. The new id for the constructed node will
make all trees different with each other. Identities, (12),
(13) are added because of this. Similarly, Aggregate-
Function and Join add their own nodes and can eliminate
duplicates – they also produce similar identities to (12),
(13), not shown here due to space limitations.

Duplicate-Elimination can destroy order, since it resorts
the input collection by the elimination key dep to remove
duplicates. Yet, if the input is already sorted by the key dep,
then it is maintained. Sort does not create additional di-
crates and it does not remove existing ones. So identities
(14) and (15) are produced.

4.4 Producing the correct output order

A bulk algebra can be extended to produce the correct or-
der using our approach. We present the sketch of an algo-
rithm that does that by extending TLC for ordering. We call
the new correct ordered algorithm TLC-C. We believe our
basic principles can be adjusted to fit any XML solution.
Pseudocode for the extension algorithm is shown in Fig. 6.
We will try to discuss the intuition behind the algorithm
using an example.

Example 4.1 Consider the query in Fig. 3. This query is
simplified on purpose to allow focus on how binding order
is dealt with. The query asks for all books sorted by the
combined document order of author, editor, hobby
(of editor) and interest (of author).

First, we present how an existing tree algebra[3] would
handle such a query, the output corresponds to the left plan

\[
\begin{align*}
1 & \text{O} \text{of} \{S[\text{any, any}(\text{)}] \Rightarrow S[\text{any, ol}(\text{)}] \}, \\
2 & \text{O} \text{of} \{S[\text{maintain}(\text{)}] \Rightarrow S[\text{maintain}](\text{O} \text{of}(\text{)}), \\
3 & \text{O} \text{of} \{J[\text{maintain}(S_1, S_2)] \Rightarrow J[\text{maintain}](\text{O} \text{of}(S_1), \text{O} \text{of}(S_2)) \}, \text{root} \notin \text{ol} \\
4 & \text{DE} \text{dl}(P[nl](\text{)}) \Rightarrow P[nl](\text{DE} \text{dl}(\text{)}), \text{if } dl \subseteq \text{nl} \\
5 & \text{DE} \text{dl}(P[nl](\text{)}) \Rightarrow P[nl](\text{DE} \text{dl}(\text{)}), \text{if } dl \subseteq \text{nl} \\
6 & \text{O} \text{of} \{C[\text{c}(\text{)}] \} \Rightarrow C[\text{c}(\text{O} \text{of}(\text{)}), \text{root} \notin \text{ol} \\
7 & \text{DE} \text{any}(P[\text{any}(\text{)}] \Rightarrow P[\text{any}](\text{DE} \text{any}(\text{)}), \\
8 & \text{O} \text{of} \{C[\text{c}(\text{)}] \} \Rightarrow C[\text{c}(\text{O} \text{of}(\text{)}), \\
9 & \text{O} \text{of} \{P[nl](\text{)}] \Rightarrow P[nl](\text{O} \text{of}(\text{)}), \text{if } ol \subseteq \text{nl} \\
10 & \text{DE} \text{any}(P[\text{any}(\text{)}] \Rightarrow P[\text{any}](\text{DE} \text{any}(\text{)}), \\
11 & \text{O} \text{of} \{R[\text{any}(\text{)}] \} \Rightarrow R[\text{any}](\text{DE} \text{any}(\text{)}), \\
12 & \text{DE} \text{tree}(C[\text{c}(\text{)}]) \Rightarrow C[\text{c}(\text{)}, \text{if } c \text{ contains newly constructed nodes} \\
13 & \text{DE} \text{dl}(C[\text{c}(\text{)}]) \Rightarrow C[\text{c}(\text{DE} \text{dl}(\text{)}), \text{if } dl \subseteq c \\
14 & \text{DE} \text{dep}(P[\text{any}(\text{)}]) \Rightarrow \text{DE} \text{dep}(\text{O} \text{of}(\text{)}), \\
15 & \text{DE} \text{dep}(P[\text{any}(\text{)}]) \Rightarrow \text{DE} \text{dep}(\text{)}
\end{align*}
\]

Table 1: Algebraic Identities used for rewrites.

Figure 6: Algorithm TLC-C for correct output order.

\text{Algorithm TLC-C}

\text{Input: a FLWOR expression Output: a TLC-C algebra plan}

\text{Globals OPERATORS ORDERLIST}

\text{procedure SingleBlock(in FLWOR) \{}

\text{Parse FLWOR, create Reductions for each Grammar Rule}

\text{For each Reduction do}

\text{Case ForClause := \{ FOR \text{ vvar in SP (Simple Path)}

\text{*Start processing as in TLC}

\text{create LC_f point SPleafnode}

\text{add LC_f to ORDERLIST}

\text{Process LET and WHERE as in TLC}

\text{Case OrderClause := ORDER BY SP_1, \ldots, SP_n, Mode}

\text{for each SP, create aftSp_i = SPtoAPT(SP_i, ”c”)}

\text{for each aftSp, create a Select S[aftSp_i] add to OPERATORS}

\text{for each aftSp, create LC, point to aftSp_i leafnode}

\text{Create Sort(value(LC_1), \ldots, value(LC_n)), add to OPERATORS}

\text{Case ReturnClause := RETURN ReturnExp}

\text{*Start processing as in TLC}

\text{if Sort was added by an OrderClause continue else}

\text{Child_{j,f} = Construct, getChild() for each LC_f found in ORDERLIST}

\text{Call Survive(Child_{j,f}, LC_j)}

\text{Create Sort(o_{j,f} = Sort(nodeid(LC_1), \ldots, nodeid(LC_n))}

\text{Construct, SetChild(SortLC_f) and SortLC_f.SetChild(Child_{j,f})}

\text{\}}</\text{function SPtoAPT(SP, mSpec)}

\text{return an APT (Annotated Pattern Tree) from SP (Simple Path)}

\text{use Rel from StepAxis of SP, use mSpec for all edges}

\text{function Survive(Operator, LC_j) \{}

\text{if Operator removes nodes from each tree (e.g. Project type)}

\text{add LC_j not in Projection List}

\text{Use edge “-” for FOR, edge “*” for LET and RETURN}

\text{Call Survive on join values and inner construct elements}

\text{if nesting on FOR and “outer” does not have OrderClause}

\text{Create LC_{j+1} for root of “inner” tree}

\text{add LC_{j+1} to generated Sort of “outer” plan}

\text{Call Survive(SortOuter, getChild), LC_{j+1})}

\text{\}}</\text{function Survive(Operator, LC_j)}

\text{procedure NestedQuery(in FLWOR) \{}

\text{Process FLWOR, if (Nested) then Call SingleBlock for “inner” and “outer”}

\text{Add a join between the outer and inner plan}

\text{Use edge “-” for FOR, edge “*” for LET and RETURN}

\text{Call Survive on join values and inner construct elements}

\text{if nesting on FOR and “outer” does not have OrderClause}

\text{Create LC_{j+1} for root of “inner” tree}

\text{add LC_{j+1} to generated Sort of “outer” plan}

\text{Call Survive(SortOuter, getChild), LC_{j+1})}

\text{\}}</\text{function NestedQuery(in FLWOR)}

\text{In Fig. 8, the first FOR clause is processed – the Simple}

\text{Part} (SP) \text{(doc/\text{book}) is converted into an Annotated}

\text{Pattern Tree (APT) with “*” edges and a variable is bound}

\text{pointing to the leaf (book node). Processing of subsequent}

\text{FOR clauses detects the existing variables. So now each SP}

\text{is merged with the existing APT. After all the FOR clauses}

\text{are processed, Select1 is generated. Next the RETURN}

\text{clause is processed and Project2 is generated to maintain}

\text{the book before it is passed to Construct3 that generates}

\text{the final output.}

The problem is an assumption that the pattern tree match
will generate the appropriate order. Unfortunately that as-
sumption is not correct. The order of the pattern tree match
is that of an in-order traversal – that would produce books
ordered by author, interest, editor, hobby, in the
example above. Clearly that is not the correct output
order. The TLC-C extension algorithm is designed to en-
force such algebras and guarantee correctness.

Example 4.2 The TLC-C output corresponds to the right
plan in Fig. 3. Each FOR clause is processed as with TLC,
but now a global ORDERLIST is also generated. The list
contains pointers to the Logical Classes for each leaf node
in the Simple Paths of every FOR clause. For the query
shown in Fig. 3, ORDERLIST would be (2), (3), (5), (6), (4).

Since no explicit ORDER BY clause is used, TLC-C creates

\text{XPath expression without branching predicate.}
5 Performance Benefits

Given the algebraic framework established above, we can develop execution plans for given query expressions. In this section, we discuss some of the optimization opportunities that become possible on account of the new algebra.

5.1 Duplicate Elimination and Efficiency

To solve the issue with no duplicates in XQuery, existing tree-based algebraic solutions [8, 3] followed the relational model and chose to operate on duplicate-free sets of trees. It is very common for operations to generate duplicates in a tree algebra (e.g., a pattern tree match followed by a projection). If the set semantics were to be maintained at all times, a duplicate elimination is required after each operator is applied. For the simple query shown in Fig. 9, such a plan corresponds to the left plan shown in Fig. 10.

Existing tree algebra implementations use (unique) node identifiers to quickly check for duplicates. Unfortunately, even though the comparison operation is so cheap, Duplicate Elimination remains a blocking procedure and its cost on large sets is significant. Forcing it after every operation can cause a major performance hit.

The semantics for XQuery require the absence of duplicates, so some form of duplicate elimination is unavoidable. But, what is the minimum that can produce the same correct results as if sets were used for each operation? The complete answer to this problem can be seen in Fig. 11. Here we will describe it intuitively with an example.

Example 5.1 We use the query shown in Fig. 9 and we show how to generate the middle plan in Fig. 10. The algorithm iterates over the existing tree algebra plan and considers each operator. Select1 and Aggregate2 add new nodes to each tree so their output is set to D-Spec = ‘tree’. Filter3 maintains the ‘tree’ D-Spec. Then Project4 removes some nodes and sets the D-Spec to ‘empty’. A Duplicate Elimination5 procedure is necessary to remove potentially redundant trees and set the D-Spec to ‘tree’. The succeeding Select6 and Construct7 do not create additional duplicates and set the output D-Spec to ‘tree’.

The produced duplicate-free result is the same as the existing algebra plan that uses sets, but now only one Duplicate Elimination is needed. The key is to change the operator input/output semantics from sets to our Hybrid Collections with Duplicate Specification (D-Spec). It enables the simple procedure, described in Fig. 11, to detect duplicates and eliminate them on-demand, instead of proactively.

The basic building block for XQuery is the FLWOR statement. The absence of duplicates in the final output of a FLWOR are essentially described by the FOR variable bindings. In a tree algebra, this means that only part of the output trees (the one that corresponds to the FOR bindings) needs to be used to produce the correct duplicate-free output. Existing tree algebras (including TLC) used a Project operation to convert into the necessary part of the tree before trying to remove duplicates. That created potential scenarios where information was projected out, only to be added later. The Duplicate Specification D-Spec we spec-
procedure FindMinimumDuplicates(in TLC-C plan)
For each Operator OP in the plan
if OP does not modify each tree structure
use D-Spec of input collection in the output
if OP modifies each tree structure - adds new nodes
use output D-Spec = 'tree'
if OP modifies each tree structure - removes nodes
use output D-Spec = 'empty'
add Duplicate Elimination DE('tree')

Figure 11: A procedure that determines the minimum Duplicate Elimination necessary for a query.

procedure partialDuplicates(in TLC-C plan)
For each Operator OP in the plan
if OP does not modify each tree structure
use D-Spec of input collection in the output
if OP modifies each tree structure, adds new nodes
if D-Spec of input is 'tree' or 'empty'
use output D-Spec = 'tree'
if D-Spec of input is a node list (dl)
if new node is a data node and ...
... connects to tree with a '-' or '?' edge
add Logical Class of new node to dl
set output D-Spec = dl
if OP modifies each tree structure, removes nodes
if D-Spec of input is 'tree' or 'empty'
use output D-Spec = 'empty'
add Duplicate Elimination DE('tree')
if D-Spec of input is a node list (dl)
if the removed nodes exist in dl
set output D-Spec = dl
else set output D-Spec = dl

Figure 12: Taking advantage of Partial Duplicates.

Example 5.2 Consider the query in Fig.5 and the middle plan in Fig.14. The query asks for the output to be duplicate-free on open_auction. We iterate over the existing plan and consider each operator. Select1 produces output in D-Spec = (2). Aggregate2 adds a new constructed node, not data nodes - so the input D-Spec is passed to the output. Filter3 maintains the D-Spec. Project4 removes some nodes but maintains (2), so it retains D-Spec = (2). No Duplicate Elimination is required, the collection has the correct D-Spec (2) that corresponds to open_auction. The succeeding Select and Construct do not create duplicates and do not modify the D-Spec, so and the final output is correct.

Note how the Project is not required for correctness any more. The right plan in Fig.14 shows the outcome of this algorithm for the query in Fig.5. Duplicate Elimination has been removed completely via the careful consideration of partial duplicates.

5.2 Selections and Ordering
The algorithm we used in Section 4.3 places a Sort operation at the end of a query plan to produce the correct output order. This placement dictates the physical implementation, which limits severely the choices for optimization. We want to overcome this problem and produce flexible plans.

The Select operator described in Section 4.3 accepts an optional ordering parameter ord that can sort the produced trees according to it. There are well known techniques that can take advantage of such parameter in Select during the physical evaluation phase. They compute tree expressions via structural joins [2], holistic joins [1] and optimal join ordering [21, 22]. These techniques provide more efficient solutions than dictating the placement of the sorting operation at the end of the query.

To take advantage of them we constructed a rewrite method that pushes the sorting operation down in a query plan using algebraic identities. This method can be applied to all selection type of queries that are based on a single block FLWOR statement without a value join. Such selection-type queries are very popular [4]. We show pseudocode for the rewrite algorithm in Fig.13. We explain it intuitively using an example.

Example 5.3 Consider the query we used in Fig.6. In Fig.14, we can see the TLC-C plan on the left and the rewritten plan on the right. The rewrite first swaps Sort3 with Project2 using identity (6) (from Section 4.3). Then it merges Sort3 into Select1 using identity (1). The ordering parameter ord specifies the order for the query now. As a side effect, the extra nodes from Project2 can be removed.

This rewrite should always be applied on single block selection queries because it cannot lead to a 'bad' plan. The selectivity of such query is determined by the original selection and the trees produced when matching the pattern tree of the selection to the database. The worst case scenario would be for the optimizer to not incorporate sorting into the pattern tree match and apply it afterwards. Since the selectivity is the same, we will do no worse than the TLC-C plan.

procedure PushSortIntoSelect(in TLC-C plan)
Find the Sort at the end of the plan
Set childOP = Sort.getChild()
while childOP ≠ Select do
if OP does not modify order
swap with Sort, use proper identity
childOP = childOP.getChild()
else if childOP modifies order
if childOP D-Spec is subset of Sort O-Spec
swap with Sort, use proper identity
childOP = childOP.getChild()
else stop, operator cannot be switched
When Select found, use ordering parameter from Sort
Remove Sort from plan

Figure 13: Merging Sort and Select.

Figure 14: On the left the TLC-C plan and on the right the rewritten plan having pushed the Sort into the Select.
Very often selection queries can include value joins and with the proposed rewrite we may miss some optimization opportunities. For this purpose, we designed the Join operator to accept an ordering parameter ord similar in concept with Select. It is applied when the join is performed and the output is ordered by it. So we can use a similar algorithm to push the Sort operation into a Join. The pseudocode is omitted both for presentation simplicity and space limitations. Instead we will show an intuitive example and discuss issues that arise.

**Example 5.4** Consider the top query from Fig.15. Fig.15 shows the plan for this query. The TLC-C plan, before the rewrite, would include the gray Sort6 operation. The Sort6 is swapped with DE5 using identity (14) and with Project4 using identity (6). Then merged to Join3 using identity (3). The ord for Join3 describes the query order now.

Yet, this is not the only alternative. The rewrite can instead use identity (4) and push the Sort further down using partial ordering. That could produce a plan where Join maintains the order (with Join3-ord = maintain(left, right)) and the Selects specify the document order (with Select1-ord = ID(2) and Select2-ord = ID(4)). Other choices include using value ordering for a more efficient merge join algorithm. Join can establish binding order (with Join3-ord = [ID(2), ID(4)]) and the Selects provide the input sorted on join value (with Select1-ord = value(2) and Select2-ord = value(4)). Or a mix of value and document order (with Join3-ord = [maintain(left), ID(4)], Select1-ord = ID(2) and Select2-ord = value(4)). The supported partial order by the Ordering Specification provides many possibilities.

Pushing Sort to Select was a much simpler rewrite than Sort to Join. The difference is due to the selectivity estimation for the query. Previously, only the selection (essentially the pattern tree match) provided the selectivity for the query. Now the optimizer has to carefully consider the cost for the two Selects and the Join before making a choice.

### 5.3 Nested FLWOR queries

Nested FLWOR statements, as we described in Section 4.4, are treated as two single block queries— one for ‘inner’ and one for ‘outer’. A Sort is used on both to establish proper order. A Join is used to connect the produced plans for ‘inner’ and ‘outer’ part. Since the order of the ‘inner’ plan is important the connecting Join is set to maintain order. This choice dictates the physical implementation for sorting and essentially forces a nested-loops join to be used.

When the nesting occurs in FOR we can overcome this problem by pulling the ‘inner’ Sort in the ‘outer’ part and over the Join, using identity (4) from Section 4.3. Now we can apply the rewrite we described earlier with pushing Sort to Join. Yet, this does not apply to the more popular choices for nested FLWOR statements—nesting in the LET/RETURN clauses. For each LET/RETURN clause, the Join needs to cluster the matching trees of the ‘inner’ plan together, hence identity (4) cannot be applied. There is no ‘outer’ Sort operation that can describe the ‘inner’ order properly. So we constructed a rewrite that uses Reorder.

The Reorder operator processes one tree at a time, reordering sub-trees as specified by its input parameters. The rewrite, that uses Reorder to provide alternative plans for nested FLWOR statements, has a very complex algorithm. The complexity is due to the variety and complexity of the nested queries it can handle. The algorithm is, for the most part, mundane and very specific to our system. Instead of presenting it in detail, we will show an example that intuitively demonstrates its behavior.

**Example 5.5** Consider the FLWOR statement nested on LET shown in Fig.16. The TLC-C plan for this query is
shown in Fig.[2] when the grayed out Sort6 operator and Join3-ord parameter are used instead of Reorder8. To produce the TLC-C plan, the ‘inner’ and ‘outer’ parts of the query are processed separately and a Sort is generated for each one. Then Join3 is used to connect them. Notice how the APT for Join3 uses the ‘*’ annotated edge to capture the LET semantics and how the ‘inner’ join node (LC=6) is added to the Project4. Duplicate Elimination5 and Construct7 operators to properly reach the connecting join. The rewrite examines the Sort6 operator of the ‘inner’ plan. Then it pulls it into the ‘outer’ plan and converts it to the appropriate Reorder8 operator. It also removes the order maintenance restriction from Join3 and enables the order of the ‘inner’ query to be anything.

The basic principle of the ‘Reorder’ rewrite is to pull the ‘inner’ Sort to the ‘outer’ part of the query. Then use Reorder to describe it and remove the order maintaining restriction from the connecting Join. Once that restriction is lifted from the Join we can produce a series of alternative plans following the techniques presented earlier (push-Sort-in-Join). The Reorder operator is non blocking (processes one tree at a time), hence it is cheaper than the ‘inner’ Sort. When used in conjunction with a join procedure, it lifts the order maintaining restrictions. So it is expected to provide a big performance increase against a blocking Sort followed by a nested loops Join.

6 Experimental Evaluation

The experiments were executed on the TIMBER [10] native XML database system. For our dataset we used the XMark [3] generated documents. Factor 1 produces an XML document that occupies approximately 707MB (472MB for data plus 241MB for indices) when stored in the database. Experiments were executed on a Pentium III-M 866MHz machine with a regular IDE hard disk and 512MB of RAM. The database was set to use an 128MB buffer pool. All numbers reported are the average of the query execution time over five executions. XQuery queries were translated to TLC-C plans using our algorithm as specified in Section 4.3. The plans utilize our algebraic operators, with no structural rewriting or cost-based optimization performed. We used an index on element tag name for all the queries, which returns the node identifiers given a tag name. On all queries that had a condition on content we used a value index, which returns the node ids given a content value. We did not use value join indices.

We executed a series of queries, those described in the XMark benchmark as well as our own. We summarize our results in Table 2. We refer to XMark queries as x13, x8 etc. Since XMark does not have single block join queries, we constructed two such queries (q1, q2) and include their results. There is nothing special about them, but due to space limitations we will not describe them here.

The results for our tests are summarized in Table 2. Our tests focus on the suggested rewrites on duplicates and ordering we presented in Section 5. The TLC-C column uses the correct plan produced as in Section 4.4. The TLC-D column uses plans produced by applying only the duplicate related rewrites (no ordering consideration). Similarly TLC-O column is for only the ordering related rewrites. Finally, TLC-OD column shows results when using both ordering and duplicate rewrites. We did not want to test TLC-C against another algebraic solution because it is not clear who address ordering properly.

Efficiency and Duplicates: In column TLC-D, we show the performance gained with the duplicate related rewrites described in Section 5.1. The rewrites can be applied to any type of query and successfully minimize the presence of Duplicate Eliminations in the plan. We always expect some performance increase over TLC-C when using this rewrite. For single block selections, we expect the performance increase to depend on the query selectivity. When many results are produced removing the blocking operation can lead to a higher performance increase – e.g. the speedup of x19 vs. x17. For single block joins and for nested queries, other factors usually dominate the cost, so the performance increase can be lower. For example, in query q2 the cost is dominated by the data materialization needed for the join and the speedup is relatively low. Similar behavior is seen in nested queries, like x8.

‘Push Sort in Select’: We tested the efficiency of our rewrite that pushes Sorts into Selects, as described in Section 5.2. The rewrite applies only to single block selection queries. Results for such queries are shown in column TLC-O for the first group of queries (x1-x20) in Table 2. Selection queries are popular in XML (the majority of the XMark queries fit the category). We expect to always see a performance increase with our rewrite on such queries. This happens because the selectivity of the query is mostly given by the Select operator. So when we merge sorting into it, we create a controlled environment for the optimizer to use well known techniques that efficiently perform the pattern tree match. The overall speedup depends on the number of results in each query. A query that produces many results is hurt more by a blocking Sort and benefits more from a semi/fully pipelined pattern tree match physical evaluation. So the performance increase is higher for such queries – e.g. x19 shows a high speedup.

‘Push Sort in Join’: Pushing Sort into a Join applies to single block join queries. Results for such queries are shown in column TLC-O for the second group of queries (q1-q2). The difference between this rewrite and the ‘push-in-Select’ is the variety of produced alternative plans. This causes the selectivity estimation to be a bit trickier, since it has to use information for the value join and the selection operations. TLC-C by default enables unordered results up to the final Sort operation. So the default Join could have been planned with sort-merge before performing the rewrite. In that case, the performance increase depends on

---

3The highest and the lowest values were removed and then the average was computed

8We believe ignoring proper order by other algebras and hence missing some sort operations can be an unfair performance advantage against TLC-C. A comparison of TLC against other algebras is shown in [12].
Table 2: Execution time for XMark queries. The queries are separated into three groups, a) single block selections (x1-x20), b) single block value join (q1,q2) and c) nested FLWOR (x8-x12).

<table>
<thead>
<tr>
<th>XMark Factor</th>
<th>TLC-C</th>
<th>TLC-D</th>
<th>TLC-O</th>
<th>TLC-OD</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>0.177</td>
<td>0.148</td>
<td>0.112</td>
<td>0.092</td>
</tr>
<tr>
<td>x2</td>
<td>0.126</td>
<td>0.187</td>
<td>0.139</td>
<td>0.077</td>
</tr>
<tr>
<td>x3</td>
<td>0.660</td>
<td>0.023</td>
<td>4.792</td>
<td>2.826</td>
</tr>
<tr>
<td>x4</td>
<td>1.064</td>
<td>0.790</td>
<td>0.688</td>
<td>0.493</td>
</tr>
<tr>
<td>x5</td>
<td>0.603</td>
<td>0.669</td>
<td>0.420</td>
<td>0.313</td>
</tr>
<tr>
<td>x6</td>
<td>0.320</td>
<td>0.228</td>
<td>0.164</td>
<td>0.154</td>
</tr>
<tr>
<td>x7</td>
<td>1.738</td>
<td>1.083</td>
<td>1.021</td>
<td>0.641</td>
</tr>
<tr>
<td>x8</td>
<td>0.951</td>
<td>0.668</td>
<td>0.621</td>
<td>0.417</td>
</tr>
<tr>
<td>x9</td>
<td>6.019</td>
<td>6.385</td>
<td>4.222</td>
<td>4.390</td>
</tr>
<tr>
<td>x10</td>
<td>4.440</td>
<td>2.376</td>
<td>1.293</td>
<td>0.967</td>
</tr>
<tr>
<td>x11</td>
<td>4.114</td>
<td>3.158</td>
<td>2.503</td>
<td>1.967</td>
</tr>
<tr>
<td>x12</td>
<td>2.754</td>
<td>1.833</td>
<td>1.746</td>
<td>0.865</td>
</tr>
<tr>
<td>x13</td>
<td>0.390</td>
<td>0.266</td>
<td>0.274</td>
<td>0.177</td>
</tr>
<tr>
<td>x14</td>
<td>0.491</td>
<td>0.259</td>
<td>0.229</td>
<td>0.145</td>
</tr>
<tr>
<td>x15</td>
<td>5.341</td>
<td>3.978</td>
<td>0.308</td>
<td>1.885</td>
</tr>
<tr>
<td>x16</td>
<td>3.149</td>
<td>1.584</td>
<td>1.807</td>
<td>1.020</td>
</tr>
<tr>
<td>q1</td>
<td>9.074</td>
<td>6.443</td>
<td>5.044</td>
<td>3.392</td>
</tr>
<tr>
<td>q2</td>
<td>14.979</td>
<td>12.633</td>
<td>13.763</td>
<td>11.564</td>
</tr>
<tr>
<td>x8</td>
<td>123.208</td>
<td>114.648</td>
<td>111.138</td>
<td>9.649</td>
</tr>
<tr>
<td>x9</td>
<td>135.944</td>
<td>117.960</td>
<td>12.411</td>
<td>10.953</td>
</tr>
<tr>
<td>x10</td>
<td>180.000</td>
<td>168.231</td>
<td>160.372</td>
<td>145.372</td>
</tr>
<tr>
<td>x11</td>
<td>155.148</td>
<td>137.014</td>
<td>16.390</td>
<td>13.443</td>
</tr>
<tr>
<td>x12</td>
<td>70.167</td>
<td>61.966</td>
<td>7.412</td>
<td>6.161</td>
</tr>
</tbody>
</table>

and Duplicate Elimination are blocking operations and can cause poor performance on big collections with many trees. Larger documents will generate results that use such collections, so we expect to see higher speedup for them. This is verified in the figure by the increasing gap between the TLC-C (top) and the other curves.

7 Related Work

In the relational world, smart sort placement has been considered an interesting problem for a long time (see [8, 13]). A paper that shares similar principles with our approach is Interesting orders [18]. Their technique assumes QGM (a dataflow graph) is used to query a relational system. They use ordering properties to annotate each box in QGM and then use graph algorithms to minimize the sorts used. But they do not seem to support partial ordering or consider duplicates like we do. Also they use sequences, whereas we use the more flexible arbitrary hybrid collections. Furthermore, it is not clear how their technique can be transferred to XML. Overall, XML poses some ordering requirements that are foreign to the relational model.

There has been some work for mapping XML to a relational system [4, 16, 19]. These solutions try to address the ordering problem converting it to SQL ORDER BY clauses. This can lead to correct results for ordering on value, but there is no comprehensive study on how the binding/document order requirements can be expressed. Also, there is no discussion on optimization possibilities for ordering and duplicates.

Native navigational-based [17] approaches iterate over data nodes, thus are able to follow the XQuery iteration model step by step. Using this model one can produce the correct output order. Following this general approach, the authors in [8] present a very thorough study for ordering in XML and use a theoretical analysis to capture the ordering semantics. They do try to touch on some optimizations for navigational systems. Yet, the instance at-a-time operation leaves a lot to be desired with regards to performance and scalability of such solutions.

Native algebraic-based [9, 11, 20] implementations, and both the node [8, 20] and tree algebras [8, 11] used as their basis, have the benefit of a more flexible optimization framework. But they have their own share of problems with XML order. They try to produce the correct output by using sets or sequences of trees. Sets lose all ordering information, so the correct result cannot be guaranteed at all times. Also, sets can create performance problems because redundant sorts will be used whenever something...
needs to be ordered. Sequences can maintain the order throughout the query plan, but they are lacking a careful consideration of XQuery ordering requirements (especially binding order). A pattern tree match will not produce the correct order as we saw in Section 4.4. Performance-wise, sequences need tightly bound operators that reduce flexibility and have to consider the entire plan before producing a rewrite (making local optimization difficult). Finally, both sets and sequences assume no duplicates are present and required several duplicate elimination operations to be applied through the plan. The solution we proposed addresses all these difficulties and also provides a good optimization framework.

8 Final Words

Ordering in XML query processing is a complex, yet important procedure, with a complex specification and significant performance ramifications. We presented a solution that uses hybrid collections annotated with an Ordering Specification as the means for a correct and flexible solution. Paired with ordering is the presence of duplicates, which we address in a similar manner adding a Duplicate Specification to our collections. We showed how an XML tree algebra can be extended to incorporate our hybrid collections producing the correct results. We presented several algebraic optimizations that take advantage of hybrid collections and experimentally demonstrated the performance increase they engender.

While the presentation in this paper was of necessity focused on a specific algebra, the concept of a hybrid collection is applicable much more generally. It can certainly be used in conjunction with any other XML algebra. In fact, hybrid collections may turn out to be useful even beyond XML. Figuring out where and how, is left to future work.

References


