A Unified Tuple-Based Algebra for XQuery

Ioana Manolescu
INRIA Futurs, Gemo group, France
Ioana.Manolescu@inria.fr

Yannis Papakonstantinou
Computer Science and Engineering
UC San Diego, USA
yannis@cs.ucsd.edu

1. INTRODUCTION

The emergence of XML and XQuery motivated many academic and industrial XML query processing and XQuery (in particular) works. Many of the works originating from the database community based XQuery processing on tuple-based algebras. The trend is hardly surprising: In the past, tuple-based algebras delivered great benefits to relational query processing but also provided a solid base for the processing of nested OQL data and OQL queries. Tuple-based algebras for XQuery carry over to XQuery key query processing benefits such as performing joins using set-at-a-time operations.

Since XQuery processing is a relatively young field, stable paradigms for query processing optimization have not emerged yet. The result is the development of multiple similar, yet different, frameworks. This is a serious problem to the communication of results and the progress of the state-of-the-art. As it has been observed in other disciplines as well and is well explained in the classic of the history of science [13], the preparadigmatic stage of a field is characterized by multiple works moving in parallel and in competition, as opposed to building upon each other. Paradigm unification eventually resolves this problem. We note that the world of relational query processing benefits from a number of clear paradigms (relational algebra, relational calculus, a well defined set of operators and implementations thereof) that facilitate education and research. While SQL, being a real language, had plenty of features that did not reduce to neat SPJ queries, it is unquestionable that clear paradigms such as SPJ (conjunctive) queries were a valid and fruitful research vehicle.

XQuery processing has not reached the point where widely acceptable paradigms for it are developed. Tuple-based algebras for XQuery are no exception to the rule.[1]

[1]The emergence of relational databases out of Codd's papers, which provided a clear and concise paradigm, are a surprising exception to the rule.

Proceedings of the Second International Workshop on *"XQuery Implementation, Experience, and Perspectives" (XIME-P)*

Our primary goal in this paper is the specification of a unified tuple-based algebra for XQuery, which removes the surface differences of prior works and shows their fundamental connections.

A secondary goal (albeit necessary for achieving the primary goal) is the reconciliation of the abstract labeled tree data model, which has been the favorite of research works with the extensive XQuery Data Model (XQDM) specification, which is what industrial implementations have to subscribe to. While it is unrealistic to expect research works to be developed on an unusually complex data model specification, it is realistic to create a clear correspondence between labeled tree abstractions and XQDM, so that both researchers and practitioners have a clear understanding of what is not being supported by each research work. To achieve the goal we take a modular approach to issues. For example, we include types in our abstraction in a way that captures both the static and dynamic typing aspects, but we assume that the type assignment is achieved by a separate module that adheres to XQuery's typing rules.

We make next a few points about the scope of this work. First, we do not address the large area of implementations of the operators described. Depending on the case (XML database, XML middleware, and so on) different implementations have been known to deliver benefits. Second, we not provide "yet another algebra". Instead, we re-use features that have already appeared in the literature and explain how features and operators that have been independently proposed correspond to each other. Third, the algebra has redundancy, which we highlight. The redundancy is justified by the fact that various works have used alternative forms of essentially similar structures because they were more suitable to their particular problem. Fourth, due to the obvious space limitations this version does not have the full detail and formalization that the full version needs to have. Finally, we do not discuss non-tuple-based implementations despite the fact that non-tuple-based implementations seem to be more efficient in particular settings.

Remark on the Motivation The two goals for this work came from the stimulating discussions that took place during XIME-P 2004 at Paris (see report at [14]). Panelists (and especially academics) stressed the lack of clean and unified concepts that can facilitate education

```
<customerList>
  <customer>
    <name> <first>John</first>
           <last>Smith</last> </name>
    <address>11 Maple</address>
  </customer>
  <customer>
    <name> <first>Mary</first>
           <last>Jones</last> </name>
    <address>456 Oak</address>
    <address>789 Pine</address>
  </customer>
  <customer>
    <name> <first>David</first>
           <last>Johnson</last> </name>
   </customer>
</customerList>
```

Figure 1: Running example XML document.

and comparison of research works.

Organization This paper is organized as follows. Section 2 presents an abstract view of XQuery's data model, on which this work is based. Section 3 extends this model with support for collections, and presents the *unified algebra* operators. Section 4 shows how the unified algebra maps to existing proposals. We then conclude.

2. ABSTRACTING THE XQUERY DATA MODEL

We present *XQDMA*, an abstract view of the XQuery data model [20] on which this algebra relies.

XQDMA comprises the basic elements of the XQuery Data Model: nodes, values, and lists [20]. Nodes correspond to subtrees of a document, and have unique identities. The result of the id-based comparison $n_1 =_{id} n_2$ is true only if n_1 and n_2 are exactly the same node. A total *document order* is established over all nodes manipulated by a query; this order is produced from the order of nodes in the input documents. Values can be compared within a specific atomic values domain; we denote such comparisons by op_v where op is a comparison operator, e.g., =. Nodes can be compared by value, by converting each node to a value, and comparing them via op_v. We extend the op_v notation to node value-based comparison.

Lists may contain nodes and/or values, possibly at several positions in the list. Lists can be compared: (i) By deep equality, which we denote $=_l$, whereas: two lists are equal if they have the same number of items, and their items are pairwise equal via $=_v$; (ii) Existentially, via $op_\exists$: the comparison succeeds if at least one element from the first list compares true via op_v to an element in the second list. We adopt the list union and intersection semantics from [21].

Whether an XML Schema of the input documents is available or not, some simple or complex type information is always attached to data model instances, whether nodes, values, or lists, manipulated by a query [21]. Unlike previous XML query languages, XQuery semantics is strongly influenced by types: **(1)** The query processor may reject ill-typed queries. **(2)** The result of a value-based comparison among nodes, values and/or lists depends on the operands' types, and the types to which they will be cast prior to the comparison [21]. **(3)** Several XQuery constructs, such as instance of, typeswitch, cast, explicitly manipulate expression types. Thus, a realistic algebra must take the type system into account.

We assume the existence of a set of valid XQDMA element and attribute names and of a set of valid XQDMA values. We assume that attribute names always start with the @ character. We view a document d as a tuple $(N_d, C_d, \lambda, \tau)$, where:

- N_d is the set of nodes in document d, which is partitioned into the following four sets of nodes: the document root node r, the set of element nodes E_d, the set of attribute nodes A_d, and the set of value nodes V_d. All nodes have identity.
- C_d is the set of edges between nodes of N_d. Following [20], N_d and C_d observe a set of conditions:
  - N_d and C_d form a tree structure.
  - The document root r has a single child which is an element node.
  - An element node $n \in E_d$ may have zero or more children belonging to $E_d \cup A_d \cup V_d$.
  - Nodes in V_d cannot have children.
  - An attribute node has exactly one child which belongs to V_d.
  - There is an order $\ll$ relationship between the children of a node, such that the attribute children precede the value and element children. The $\ll$ relationship between attribute nodes is always undefined. Thus, $\ll$ between element nodes, value nodes and combinations thereof may or may not be defined.
- λ is a node labeling function, where the label $\lambda(n)$ of node n is:
  - a XQDMA element name, if the node n belongs to E_d or A_d.
  - a typed value v from the set of XQDMA values, if the node n is a value node.
- τ is a function assigning to any node $n \in N_d$ a type $\tau(n) \in \mathcal{T}$, where $\mathcal{T}$ is the set of value and node types considered in XQuery's type system [20, 22]. The types assigned by τ are consistent with the dynamic typing reasoning from [22].

In the sequel, we will consider only queries for which static type analysis either was not attempted, or it succeeded, and we will rely on τ whenever necessary, namely, for op_v comparisons, and runtime access to type. We will term any instance of the above abstract data model an *XQDMA instance*; similarly, an *XQDMA type* is a type from $\mathcal{T}$ as defined in [20, 22]. We will also assume available a *node creation function* ν, which produces new nodes with fresh identity, given the node's possible children, and label. We denote by $tree(n)$ the entire tree rooted at node n.

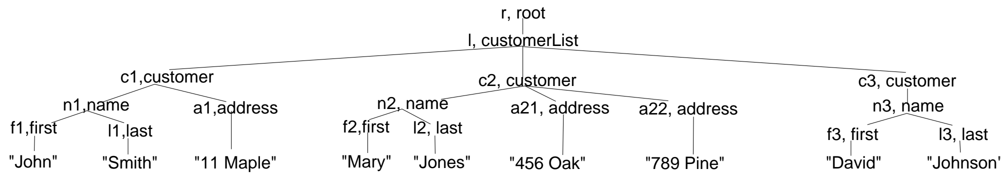

Figure 2: Labeled tree data representation of the sample document.

3. A UNIFIED TUPLE-BASED XQUERY ALGEBRA

The last years have seen many tuple-based algebras being designed and used in XQuery processors. The *unified algebra* presented next is meant to establish a common ground among such algebras, while at the same time making an effort to get closer to the formal semantics of XQuery.

The unified algebra is based on an extension of XQDMA with sets, bags, and lists of tuples; notice that the extensions are only visible internally, to algebra operators. We describe first the *Unified Data Model*, then proceeed to describing the algebra.

3.1 Unified Data Model

We extend XQDMA with the following types.

Tuples have the structure $[\$a_1 = val_1, \ldots, \$a_k = val_k]$, where each $\$a_i = val_i$ is a *variable-value* pair. Variable names such as $\$a_1$, $\$a_2$ etc. follow the syntactic restrictions associated to XQuery variable names; they often (but not always) correspond to query variables. Variable names are unique within a tuple. A value may be (i) the special constant $\perp$ (*null*),[2] (ii) a node or value as described in the previous section, or (iii) a (nested) set, list, or bag of tuples. Given a tuple $[\$a_1 = val_1, \ldots, \$a_k = val_k]$ the list of names $[\$a_1, \ldots, \$a_k]$ is called the *schema* of the tuple. Notice that we allow variable values to be nested sets/bags/lists since some systems may exploit this for building efficient evaluation plans.

Beside lists, we also support sets and bags of tuples. We denote lists as $[t_1, \ldots, t_n]$, bags as $\{\!\{t_1, \ldots, t_n\}\!\}$, and sets as $\{t_1, \ldots, t_n\}$. We will refer to sets, lists, and bags collectively as *collections*. In all three cases the tuples $t_1, \ldots, t_n$ have the same schema. Sets have no duplicate tuples, i.e., no two tuples in a set are *id-equal* as defined below. Bags and lists may have duplicate tuples.

Two tuples are equal, denoted as $t_1 =_{id} t_2$, if they have the same schema and the values of the corresponding variables either (i) are both null, or (ii) are both values and compare equal via $=_v$, or are both nodes and compare equal via $=_{id}$, or (iii) are both sets of tuples and each tuple of a set is id-equal to a tuple of the other set. For the case (iii), similar definitions apply if the variable values are bags or lists of tuples. In particular, in the case of bags, we also take into account the multiplicity of each tuple and, in the case of lists, the order of the tuples.

[2]The XQuery Data Model also has a concept of *nil*. We do not merge the two concepts, in order to avoid the discrepancies that may stem from the overloading.

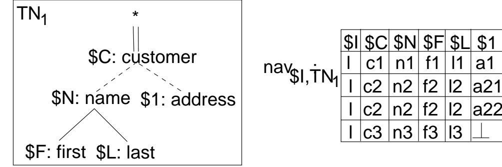

Figure 3: Tree pattern for XQuery Q1.

Notation Given a tuple $t = [\ldots \$x = v \ldots]$ we say that $\$x$ maps to to v in the context of t. We represent by $t.\$x$ the value that the variable $\$x$ maps to in the tuple t. The notation $t' = t + (\$var = v)$ indicates that the tuple t' contains all the variable-value pairs of t and, in addition, the variable-value pair $\$var = v$. The tuple $t'' = t + t'$ contains all the variable-value pairs of both tuples t and t'. Finally, we denote by (id) the node whose identifier is id.

Sample document and query We use for illustration the sample XML document in Figure 1. The same document is shown under a tree form in Figure 2, where each node n is designated by $(id, \lambda(n))$. For illustration, we will use the following query:

```
for $C in $I//customer                              (Q1)
return <customer>
         { for $N in $C/name
               $F in $N/first
               $L in $N/last
           return {$F, $L, $C/address } }
         <customer>
```

3.2 Unified Algebra

Current tuple-based XQuery algebras typically consist of operators that: (i) navigate into the data and deliver collections of bindings (tuples) for the variables (Section 3.3); (ii) construct XQuery Data Model values from the tuples (Section 3.4); (iii) allow for nested plans (Section 3.5); and (iv) combine collections applying operations known from the relational algebra, such as joins (Section 3.7), (v) contain some redundant operator, such as structural joins (Section 3.6) that deliver performance gains in particular environments.

Given an XQuery q with a set of free variables $\overline{V}$, we translate it into a corresponding algebraic expression f_q, which is a function whose input schema is $\overline{V}$. The algebraic expressions output either sets/bags/lists of tuples, or XQDMA values.

3.3 Navigation

Navigation operators follow the established (from XPath research) tree pattern paradigm. XQuery tuple-based algebras extend tree patterns in two main directions. First, since in a nested XQuery a navigation in the inner query may start from a variable of the outer query, we need tree patterns where the root of the navigation is a variable binding. Second, nested queries are often captured by tree patterns with "optional" subtrees: if no match is found for optional subtrees, then those subtrees are ignored and a $\perp$ (null) value is returned for variables that appear in them. This feature reminds of outerjoins, and has been used in some algebras to consolidate the navigation of multiple nested queries (and hence of multiple tree patterns) into a single one.

An unordered *tree pattern* is a labeled tree, whose nodes are labeled with (i) an XQDMA element/attribute or the wildcard $*$ and (ii) optionally, a variable $\$var$. The edges are either *child* edges, denoted by a single line, or *descendant* edges, denoted by a double line. Furthermore, edges can be *required*, shown as continuous lines, or *optional*, depicted with dashed lines.

In order to formally define the optional navigation feature, we introduce the notion of *more specific tuples.* Given two tuples t and t' with the same schema, we say that t is *more specific* than t' if for every variable $\$V$ either $t.\$V = t'.\$V$ or $t'.\$V = \perp$. For example, $[\$A = (n_a), \$B = \perp, \$C = \perp]$ is less specific than $[\$A = (n_a), \$B = (n_b), \$C = \perp]$.

Let n be a node and T be a tree pattern with the variables $\$V_1, \ldots, \$V_k$. The set of mappings $map(T, n)$ of T on n is the minimal set of tuples $[\$V_1 = v_1, \ldots, \$V_k = v_k]$, where every v_i is a node in the subtree rooted in n, and the following conditions hold. Note that we establish a mapping from the nodes of T to the nodes of $tree(n)$. The mapping will then imply a corresponding mapping from variables that appear in T to nodes of $tree(n)$ in the obvious way.

- the root of T maps to n;
- if the node n_T of T maps to the node n_t of $tree(n)$, then either n_T and n_t have the same label, or the label of n_T is $*$;
- if there is a required child (resp. descendant) edge from node n_T to n'_T of T, and nodes n_T and n'_T map to the nodes n_t and n'_t, then n_t is the parent (resp. ancestor) of n'_t.
- if there is an optional child (resp. descendent) edge between n_T and n'_T then (i) the node n'_T maps to $\perp$ or (ii) n_T and n'_T map to n_t and n'_t of $tree(n)$ and n_t is the parent (resp. an ancestor) of n'_t.
- there is no more specific tuple of mappings in $map(T, n)$.

Many existing works only considered set-driven evaluation. To keep in line with XQuery semantics, we extend to ordered tree patterns that produce lists of mappings. The above definition produces unordered tuples.

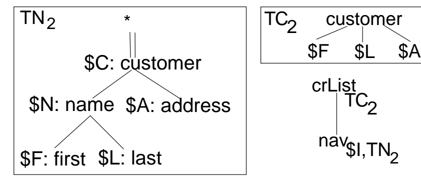

Figure 4: Navigation and construction for Q2.

We first order the variables of the result tuples by their depth-first, pre-order of appearance in the tree pattern. We then order the tuples of *map* by the lexicographic order implied by the order of the variables. We illustrate later with an example.

The *map* definition extends in the expected way to the case where the second argument is an XQDMA list as opposed to a single XQDMA node.

The navigation operator $nav_{\$R,T}$ inputs and outputs a list of tuples. The parameter $\$R$ is a variable of the input list and T is a tree pattern. The output of the operator on input $[t_1, \ldots, t_n]$, where each t_i, $i = 1, \ldots, n$ is a tuple of the form $[\$R = r_i, \ldots]$, is:

$$[t_1 + t \mid t \in map(T, r_1)] \ \# \ldots \# \ [t_n + t \mid t \in map(T, r_n)]$$

where $\#$ is the concatenation operator and the *list comprehension* notation $[t_i + t \mid t \in map(T, r_i)]$ denotes that t ranges over the list $map(T, r_i)$ and for each one of its bindings a tuple $t_i + t$ is output.

Figure 3 shows the pattern TN_1 corresponding to the navigation part in query Q1, and the result of nav_{TN_1} on our sample document. The order of the tuples is justified as follows: The depth-first pre-order of the variables in the tree pattern is $(\$C, \$N, \$F, \$L, \$1)$. Consequently, the first tuple precedes the second because $c_1 << c_2$ and the second tuple precedes the third tuple because $a_{21} << a_{22}$.

Remark: navigation axes Our navigation patterns only include the child and descendent axes, since these are by far the most popular in existing algebras and systems. In principle, they could be extended to other axes.

3.4 Construction

Tuple-based XQuery algebras include operators that construct XML from the bindings of variables. This is captured by the $crList_L$ operator, which inputs a list/set/bag of tuples and outputs a XQDMA list. The parameter L is a list of *construction tree patterns*, called a *construction pattern list (cpl).* The kinds of nodes in cpl's roughly correspond to the kinds of nodes in XQDMA. Given a tuple t and the cpl L, the instantiation $inst_L(t)$ of the cpl is the XQDMA list produced by the substitutions of cpl nodes, as follows:

Each *element construction pattern node* n_e is annotated with either (i) an XQDMA element name l or (ii) an XQDMA variable $\$V$. In $inst_L(t)$ the node n_e is substituted with a fresh element node n'_e whose label is l or the value of the variable $\$V$.

Each *content construction pattern node* n_c is always a

leaf node of the cpl and is annotated with either (i) an XQDMA variable $\$V$, or (ii) an XQDMA list l. In the latter case, in $inst_L(t)$ the node n_c is substituted by l. In the former case, in $inst_L(t)$ the node n_c is substituted by the XQDMA value of $\$V$, i.e., by a list of element, value or attribute variables. If n_c is the root node of a construction tree pattern and is substituted by element or attribute nodes, then the nodes maintain their identities. Otherwise, the nodes are assigned new identities unique within the list and across the lists produced by multiple instantiations, by calling ν.

Each *attribute construction pattern node* n_a is annotated with either (i) an XQDMA variable V, or (ii) a value s from the set of XQDMA attribute names. In $inst_L(t)$ the node n_a is substituted with a fresh element node n'_a whose tag is the string s or the value of the variable V.

Given an input list $I = [t_1, \ldots, t_n]$ the operator $crList_L(I)$ produces the list of nodes:

$$inst_L(t_1)\# \ldots \#inst_L(t_n)$$

where $\#$ is the concatenation operator.

For example, consider the query:

```
for $C in $I//customer, $N in $C/name,        (Q2)
    $F in $N/first, $L in $N/last,
    $A in $C/address
return <customer> { $F, $L, $A } </customer>
```

Figure 4 depicts the navigation pattern TN_2, the *cpl* TC_2 consisting of a single construction pattern, and a simple algebraic expression, corresponding to Q2.

3.5 Nested Plans

We have seen how the combination of navigation and construction operators captures the navigation and construction of unnested FLWR XQueries. Next we introduce two flavors of operators for handling nested queries.

Apply The $app_{p \mapsto \$R}$ (as in "*app*ly plan") unary operator has as parameter an algebra expression p, which delivers an XQDMA list, and a fresh result variable $\$R$ which does not appear in its input. Intuitively, for every tuple t of the input collection I, we evaluate $p(\{t\})$ and the result is assigned to $\$R$.

$$app_{p \mapsto R}(I) = \{t + (R = r) | t \in I, R = p(\{t\})\}$$

For example, recall query Q1, producing a `customer` output element for every `customer` element in the input. The output element includes the first name and last name pairs (if any) of the customer and for each name all the address children (if any) of the input element. Figure 5 depicts an algebraic plan for Q1, using the *app* operator. TC_3, TA_3 and TN_3 are navigation patterns, while TU_1, TU_2 and TU_3 are *cpl*s, consisting of one, two, and one respectively construction patterns. The partial plans p_1 and p_2 each apply some navigation starting from $\$C$ and construct XML output in $1, respectively,

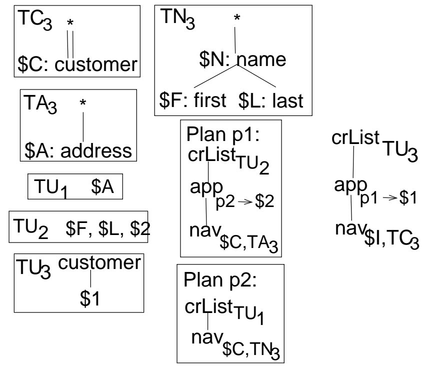

Figure 5: Algebraic plan for Q1.

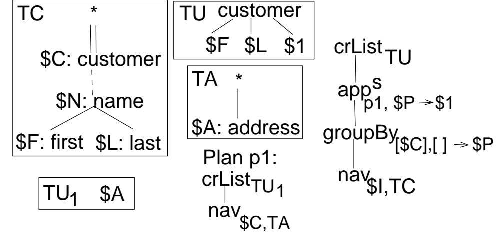

Figure 6: Alternative algebraic plan for Q1.

$2. The first *app* operator reflects the XQuery nesting, while the second *app* corresponds to the inner return clause. The final *crList* operator produces `customer` elements.

The *app* operator is defined on one tuple at a time. However, many architectures (and algebras) consider also a set-at-a-time execution. For instance, instead of evaluating $app_{p2 \mapsto \$2}$ on one tuple at a time (which means twice for customer $c2$ since he has two addresses), one could group its input by customer ID, and apply p_1 on one group of tuples at a time. In such cases, the following pair of operators is useful.

Group-By The $groupBy_{\overline{G_{id}}, \overline{G_v} \mapsto \$R}$ has three parameters: the list of group-by-id variables $\overline{G_{id}}$, the list of group-by-value variables $\overline{G_v}$, and the result variable $\$R$. The operator partitions its input I into sets of tuples $P_{\overline{G}}(I)$ such that all tuples of a partition have id-equal values for the variables of $\overline{G_{id}}$ and equal values for the variables of $\overline{G_v}$. The output consists of one tuple for each partition. Each output tuple has the variables of $\overline{G_{id}}$ and $\overline{G_v}$ and an additional variable $\$R$, whose value is the partition.[3] Note that we drop from the input any tuple that has a $\perp$ in any of the of the $\overline{G_{id}}$ or $\overline{G_v}$ variables.

Apply-on-Set The $app^s_{p, \$V \mapsto \$R}$ operator assumes that the variable $\$V$ of its input I is bound to a set/bag/list of tuples. The operator applies the plan p on $t.\$V$ for every tuple t from the input, and assigns the result to the new variable $\$R$.

[3]Some algebras do not repeat the variables of $\overline{G_{id}}$ and $\overline{G_v}$ in the partition, for efficiency reasons.

For example, Figure 6 shows another possible plan for Q1. This time we use a single navigation pattern TC, which includes optional edges. The navigation result may contain several tuples for each customer with multiple addresses. Thus, we group the navigation result by $\$C$ before applying p_1 on the sets.

The benefits of implementing nested plans by using grouping and operators that potentially deliver null tuples (outerjoin in particular) had first been observed in the context of OQL.

Remark: equivalent expressions Comparing Figure 6 with Figure 5 shows that the same query may be expressed by different expressions in the unified algebra. This is a deliberate choice, as our algebra aims at unifying and consolidating existing algebraic designs (insofar as they are compatible, and do not contradict XQuery's data model [20]), not choosing one at the expense of another. Logical structural joins, described next, further increase the space of alternatives.

3.6 Navigation Using Structural Joins

An alternative way to define the semantics of the *nav* operator is based on logical structural joins. Intuitively, a structural join combines two inputs according to the presence of a structural relationship (parent-child or ancestor-descendent) among them. Though the structural join/outerjoin brings no extra expressive power with respect to tree patterns, it enables an efficient implementation in various settings.

Without loss of generality, we will focus in the sequel on ancestor-descendent joins; parent-child joins are very similar. More generally, structural joins could be defined for structural relationships derived from other navigation axes; a *sibling(s_1, s_2)* predicate, together with $\ll$, can be used to define these [22]. We do not insist on this further.

We need the following flavors of logical structural joins to correctly capture *nav* semantics. Let R and S be two collections of tuples, such that the values of $R.\$x$ and $S.\$y$ are single nodes. For any tuple $t_R \in R$, let $desc^{ad}_{\$y}(t_R.\$x, S)$ be the set of tuples $t_S \in S$ such that $t_R.\$x$ is an ancestor (or a parent) of $t_S.\$y$. Notice that $desc^{ad}_{\$y}(t_R.\$x, S)$ may be empty.

DEFINITION 3.1. (**Ancestor-descendent structural joins**) *The structural join of R and S, on the condition that $R.\$x$ be an ancestor of $S.\$y$, is:*

$$R \bowtie^{ad}_{\$x,\$y} S \quad = \quad \{t_R + t_S \mid t_R \in R, t_S \in desc^{ad}_{\$y}(t_R.\$x, S)\}$$

The structural semijoin of R and S is defined as:

$$R \ltimes^{ad}_{\$x,\$y} S \quad = \quad \{t_R \in R \mid desc^{ad}_{\$y}(t_R.\$x, S) \neq \emptyset\}$$

The structural outerjoin of R and S is:

$$R \rhd\!\!\lhd\!\!\sqsubset^{ad}_{\$x,\$y} S = \{t_R + t_S \mid t_R \in R,\ t_S \in desc^{ad}_{\$y}(t_R.\$x, S)\} \cup \{t_R + \bot_{t_s} \mid t_R \in R, desc^{ad}_{\$y}(t_R.\$x, S) = \emptyset\}$$

In the above, $\bot_{t_S}$ denotes a tuple having t_S's schema, and all fields set to $\bot$. The definitions consider set semantics (no duplicates). For list semantics, the join result can be ordered, first according to R and then according to S order, as we did for map. A simple extension applies for bags. $\diamondsuit$

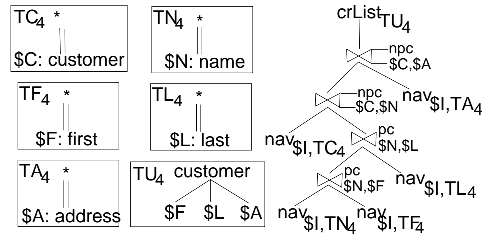

Figure 7: Alternative navigation fragment for Q1.

The above structural joins produce flat tuples, just like *nav*. Recent works have advocated the usage of pattern matching which preserves the structure among the nodes in its input, by nesting bindigs for a child node as children of the corresponding parent node binding. This is captured by nest structural join operators:

DEFINITION 3.2. (**Ancestor-descendent nest structural joins**) *The nest structural join of R and S, on the condition that $R.\$x$ is an ancestor of $S.\$y$, is:*

$$R \bowtie^{nad}_{\$x,\$y} S \quad = \quad \{t_R + (\$s = desc(t_R, S))\}$$

The nest structural outerjoin of R and S is defined as:

$$R \rhd\!\!\lhd\!\!\sqsubset^{nad}_{\$x,\$y} S \quad = \quad \{t_R + (\$s = desc(t_R)) \mid t_R \in R\}$$

These definitions are easily extended to the case when R and S are lists; the join result is ordered first by R's order, then by S's. $\diamondsuit$

Similar definitions apply to *parent-child* structural joins, denoted $\bowtie^{pc}$, $\rhd\!\!\lhd\!\!\sqsubset^{npc}$, $\ltimes^{pc}$ etc. The symetric counterpart of these structural joins, namely $\bowtie^{da}$, $\rhd\!\!\lhd\!\!\sqsubset^{nda}$, $\ltimes^{da}$, $\bowtie^{cp}$, $\rhd\!\!\lhd\!\!\sqsubset^{ncp}$, $\ltimes^{cp}$, and their nested variants, consider the ancestor (respectively, the child) to be on the left-hand side of the join.

For example, Figure 7 shows an alternative algebraic expression for Q1, based on structural joins. Notice that $\$N$, $\$F$ and $\$L$ are connected via structural joins, since they multiply each other's cardinality in the inner FLWR expression. The outer FLWR variable $\$C$ is connected via nested outer structural joins to the expressions from the inner FLWR. In Figure 7, TU_4 is a construction pattern, the other are navigation patterns.

Industrial products from IBM [4], BEA [11], Microsoft, or Natix [10], as well as prototypes such as Timber [17], indeed use (physical) structural joins to implement *nav*. Many physical structural join operators have been proposed so far in [2] and subsequent works. Alternatively, a single holistic structural join operator of [5] can be used to evaluate a complete *nav* in the absence of optional edges. Interestingly, new ID encoding schemes such as [15] allow to infer by looking at two IDs more

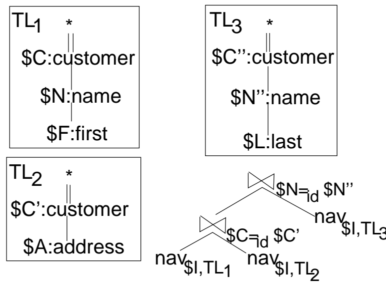

Figure 8: Join expression equivalent to $nav_{\$I,TN_3}$.

structural relationships than just descendence, thus creating opportunities for structural joins along more axes.

3.7 Relational-like Operators

In this section, we consider a set of operators related to similar (nested) relational algebraic counterparts. We use the symbols R, S etc. to refer to two sets (respectively, lists, or bags) of tuples.

We consider selections $\sigma_\theta(R)$, where θ is a boolean combination of predicates over variables of R. Furthermore, we impose that θ only applies to XQDMA variables (i.e., variables that always bind to XQDMA lists). Table 1 lists a few XQuery predicates often considered in research works, showing the notations used by this algebra, XQuery's notation, and the XQuery functions or operators backing the predicate. The semantics of predicates decorated with * is defined depending on the types of the operands.

$R \times S$ computes the cartesian product of R and S.

A θ-join $R \bowtie_\theta S$ is defined as $\sigma_\theta(R \times S)$. In the particular case where θ consists of $=_{id}$ comparisons, θ joins can be used to express a navigation operation in terms of simpler navigations, restricted to linear paths.

For example, consider the navigation pattern TN_3 from Figure 5 (ignore for now the rest of the figure). $nav_{\$I,TN_3}$ is equivalent to the join tree in Figure 8.

Projecting the variables $\$v_1, \ldots, \$v_k$ from R is achieved by $\pi_{\$v_1,\ldots,\$v_k} R$. Duplicate-eliminating projection, denoted π^0, can be achieved by grouping and not using the result of grouping. (It was well known from the relational database literature that duplicate elimination is a special grouping operator.)

4. MAPPING TUPLE-BASED XQUERY ALGEBRAS TO THE UNIFIED ALGEBRA

We briefly outline how existing XQuery algebras and similar works relate to the unified algebra exposed here.

The TAX [12] algebra is based on a simple ordered tree model, comprising nodes and values. The notion of ID used in TAX is weaker than XQuery's notion considered here; for instance, a deep copy of an element node in TAX would have the same ID as the original node. The algebra manipulates sets of trees (with extensions to bags). TAX *selection* is similar to *nav*, applying a tree pattern on a set of input trees. TAX tree patterns are similar to ours, but do not contain optional edges. TAX selections produce trees, isomorphic to the pattern tree, whereas *nav*produces tuples. Though TAX does not explicitly use tuples, it implicitly does, since all produced trees have an"homogenous" skeleton on which variable bindings are attached. Note that the usage of such intermediary trees is incompatible with XQuery's strong notion of IDs, since a node in an intermediary tree would end up having two parents, one in the original document and another in the result tree.

TAX patterns include selection predicates on nodes, captured by selections in our algebra. TAX *projection* and *product* are similar to ours. TAX *joins and outerjoins* pair two trees satisfying a join condition as the two children of a new tree. TAX allows *grouping* trees into partitions, each of which is again a tree with an artifficial root. Simple *tree update* operations are provided in TAX (e.g., node renaming, tree copy-paste, node insertion and deletion etc.). XML result construction is handled by grouping and updating result to align it with the desired return template (returned nodes do not have fresh identity as XQuery requires).

The YAT [7] algebra was devised for unordered data, and is more limited *wrt* navigation. SAL [3] was closely inspired from YAT and the OQL algebra [8]. Xstasy [18] derives from YAT and is closer to XQuery semantics. Xstasy is based on ordered trees, endowed with XQuery-style identifiers; it uses sets, and nested tuples internally, wrapped as trees (see comment on TAX's internal trees). Xstasy assumes available a global node ordering function, which is used by a *Sort* when ordered results are needed. The *path* operator corresponds to *nav*; it uses tree patterns with compulsory or optional edges. *path* may capture in a variable either *every* matching node, as in our algebra, or *all* nodes; our algebra achieves the latter by grouping on top of *nav*, or via nested structural joins. The *return* operator is similar to our *crList*; it assigns fresh identities to returned nodes, in keeping with XQuery semantics. Xstasy selections apply on nested tuples and incorporate simple predicates, existential and universal quantifiers etc. *TupJoin* is a simple join operator defined only on tuple sets. *DJoin* corresponds to our *apply*.

Improving over TAX, generalized tree patterns [6] have introduced optional edges into structural navigation patterns, and argued for the usefulness of outer structural joins. The more recente logical tree classes [17] demonstrated the interest of structure-preserving pattern matching, and introduced nest structural joins to support this.

We have taken inspiration from [11] in defining XQDMA.

The algebra presented in [16] has the features presented in our unified algebra and plans followed the pattern "collect binding with navigations and joins", "potentially apply nested plans", "package the results in XML lists". Various surface differences from our algebra were present. For example, only linear paths were captured by individual navigation operators. Multiple navigation operators were combined to achieve the effect of a tree pattern navigation operator. The proposal in [19] also features pattern matching that preserves the

Predicate	Meaning	XQuery notation / function or operator [21, 22]
$=_{id}$	ID-based equality on nodes	is / op:is-same-node
$=_v$	Value-based equality*	eq / fn:compare, op:numeric-equal etc.
$<_v$	Value-based inequality*	lt / fn:compare, op:numeric-less-than etc.
$\ll$	Order comparison on nodes	$\ll$ / op:node-before
$=_l$	Deep equality on lists*	eq / fn:deep-equal
$=_\exists$	Existential equality on lists*	= / functions backing eq on values and nodes

Table 1: Some predicates used in boolean expressions.

input structure. Their data model features collections (sets, bags, and lists), which can be nested. In contrast, XQuery's data model only supports simple lists, while our unified data model includes sets and bags, but requires structure alternation as in [1]: we allow tuples to contain lists, and lists to consist of tuples, but we do not allow tuples of tuples or lists of lists.

Group-by is a feature often considered useful in XML querying; proposals in this direction are, e.g., [19, 9, 4]. An interesting alternative (for the particular case of grouping by IDs) is provided by nested structural joins

Some operators in the unified algebra behave similarly to OQL algebras [8], but we depart from that on several aspects in order to get closer to XQuery.

5. CONCLUSIONS

We have outlined an unified tuple-based algebra for XQuery processing. The goal of this work is to bring closer existing results and efforts from academia and industry on implementing, explaining, and teaching XQuery. A key motivation for this work is the perceived need of a common ground for communication and research around the topic; such a ground will be a step forward towards "maturing" XQuery research.

6. REFERENCES

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *JCSS*, 33(3):361–393, 1986.

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

[3] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB*, 1999.

[4] K. S. Beyer, R. Cochrane, L. S. Colby, F. Ozcan, and H. Pirahesh. XQuery for analytics: Challenges and requirements. In *XIME-P*, pages 3–8, 2004.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[6] Z. Chen, H. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, 2003.

[7] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion ! In *SIGMOD*, 1998.

[8] S. Cluet and G. Moerkotte. Nested queries in object bases. Technical report, 1995.

[9] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, 2004.

[10] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.

[11] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *VLDB Journal*, 13, 2004.

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

[13] T. S. Kuhn. *The Structure of Scientific Revolutions*. 1962.

[14] I. Manolescu and Y. Papakonstantinou. Report on the first XIME-P workshop. *SIGMOD Record*, 33(4), 2004.

[15] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, 2004.

[16] Y. Papakonstantinou, V. R. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos, and P. Velikhov. XML queries and algebra in the Enosys integration platform. *Data Knowl. Eng.*, 44(3):299–322, 2003.

[17] S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for the efficient evaluation of XQuery. In *SIGMOD*, 2004.

[18] C. Sartiani and A. Albano. Yet another query algebra for XML data. In *IDEAS*, 2002.

[19] S. D. Viglas, L. Galanis, D. J. DeWitt, D. Maier, and J. F. Naughton. Putting XML query algebras into context. Available at: http://www.cs.wisc.edu/niagara/Publications.html.

[20] XQuery 1.0 and XPath 2.0 Data Model. www.w3.org/TR/xpath-datamodel.

[21] XQuery 1.0 and XPath 2.0 Functions and Operators. www.w3.org/TR/xpath-functions.

[22] XQuery 1.0 Formal Semantics. www.w3.org/TR/2005/WD-xquery-semantics.