XML TUPLE ALGEBRA

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

Vasilis Vassalos
Dept. of Informatics
Athens University of Economics and Business
Athens, Greece
vassalos@aueb.gr

SYNONYMS

Relational algebra for XML, XML algebra

DEFINITION

An XML tuple-based algebra operates on a domain that consists of sets of tuples whose attribute values are *items*, i.e., atomic values or XML elements (and hence, possibly, XML trees). Operators receive one or more sets of tuples and produce a set, list or bag of tuples of items. It is common that the algebra has special operators for converting XML inputs into instances of the domain and vice versa. XML tuple-based algebras, as is also the case with relational algebras, have been extensively used in query processing and optimization.
[1, 3, 2, 4, 6, 5, 7, 9, 10, 8]

HISTORICAL BACKGROUND

The use of tuple-based algebras for the efficient set-at-a-time processing of XQuery queries follows a typical pattern in database query processing. Relational algebras are the most typical vehicle for query optimization. Tuple-oriented algebras for object-oriented queries had also been formulated and have a close resemblance to the described XML algebras.
Note that XQuery itself as well as predecessors of it (such as Quilt) are also algebras, which include a list comprehension operator ("FOR"). Such algebras and their properties and optimization have been studied by the functional programming community. They have not been the typical optimization vehicle for database systems.

SCIENTIFIC FUNDAMENTALS

The emergence of XML and XQuery motivated many academic and industrial XML query processing works. Many of the works originating from the database community based XQuery processing on tuple-based algebras since in the past, tuple-based algebras delivered great benefits to relational query processing and also provided a solid base for the processing of nested OQL data and OQL queries. Tuple-based algebras for XQuery carry over

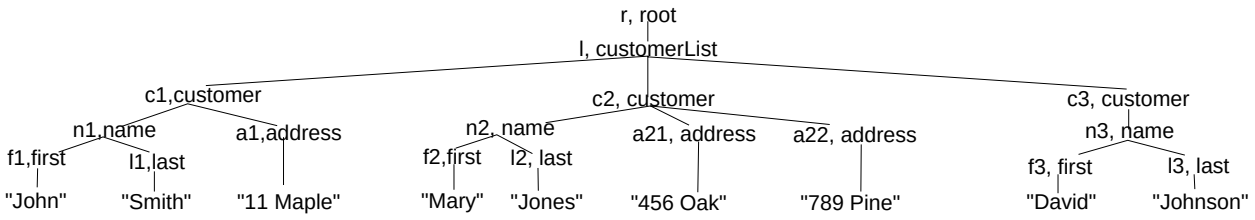

Figure 1: Tree representation of sample XML document.

to XQuery key query processing benefits such as performing joins using set-at-a-time operations.
A generic example algebra, characteristic of many algebras that have been proposed as intermediate representations for XQuery processing, is described next. It is based on a *Unified Data Model* that extends the XPath/XQuery data model [[reference to Encyclopedia article on XQuery/XQuery Data Model]] with sets, bags, and lists of tuples; notice that the extensions are only visible to algebra operators.
Given an XML algebra, important query processing challenges include efficient implementation of the operators as well as cost models for them, algebraic properties of operator sequences and algebraic rewriting techniques, and cost-based optimization of algebra expressions.

Unified Data Model

The *Unified Data Model* (UDM) is an extension of the XPath/XQuery data model with the following types:

Tuples, with structure $[\$a_1 = val_1, \ldots, \$a_k = val_k]$, where each $\$a_i = item_i$ is a *variable-value* pair. Variable names such as $\$a_1$, $\$a_2$, etc. follow the syntactic restrictions associated with XQuery variable names; Variable names are unique within a tuple. A *value* may be (*i*) the special constant $\perp$ (*null*),[1] (*ii*) an *item*, i.e., a node (generally standing for the root of an XML tree)or atomic value, 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. We restrict our model to *homogeneous* collections (sets, bags or lists) of tuples. That is, the values taken by a given variable $\$a$ in all tuples are of the same kind: either they are all items (some of which may be null), or they are all collections of tuples. Moreover, in the latter case, all the collections have the same schema.

Note that the nested sets/bags/lists are typically exploited for building efficient evaluation plans.

If the value of variable $t.\$a$ is a set, list or bag of tuples, let $[\$b_1, \$b_2, \ldots, \$b_m]$ be the schema of a tuple in $t.\$a$. In this case, for clarity, we may denote a $\$b_i$ variable, $1 \leq i \leq m$, by $\$a.\$b_i$ (concatenating the variable names, from the outermost to the innermost, and separating them by dots).

- *Lists, bags and sets of tuples*, denoted as $\langle t_1, \ldots, t_n \rangle$, $\{\!\{t_1, \ldots, t_n\}\!\}$, and $\{t_1, \ldots, t_n\}$ and referred to collectively as *collections*. In all three cases the tuples $t_1, \ldots, t_n$ must have the same schema, i.e., collections are homogeneous. Sets have no duplicate tuples, i.e., no two tuples in a set are *id-equal* as defined below.

Two tuples are id-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 equal atomic values (i.e., they compare equal via $=_v$), or are both nodes with the same id (i.e., they 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, by taking into account the multiplicity of each tuple in bags and the order of the tuples for lists.
Notation Given a tuple $t = [\ldots \$x = v \ldots]$ it is said that $\$x$ maps to v in the context of t. The value that the variable $\$x$ maps to in the tuple t is $t.\$x$. 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, (id) denotes the node with identifier id.
Sample document and query A sample XML document is shown in tree form in Figure 1. For readability, the figure also displays the tag or string content of each node. The following query will be used to illustrate the XML tuple algebra operators:

```
for $C in $I//customer                                    (Q1)
```

[1] The XQuery Data Model also has a concept of *nil*. For clarity of exposition, the two concepts should be considered as distinct, in order to avoid discrepancies that may stem from the overloading.

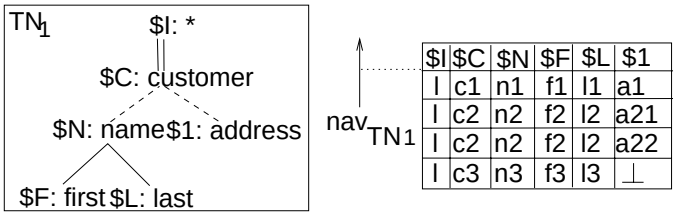

Figure 2: Tree pattern for XQuery Q1.

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

Unified Algebra

Tuple-based XQuery algebras typically consist of operators that: (*i*) perform navigation into the data and deliver collections of bindings (tuples) for the variables; (*ii*) construct XPath/XQuery Data Model values from the tuples; (*iii*) create nested collections ;(*iv*) combine collections applying operations known from the relational algebra, such as joins It is common to have redundant operators for the purpose of delivering performance gains in particular environments; structural joins are a typical example of such redundant operators.
An XQuery q, with a set of free variables $\overline{V}$, is translated into a corresponding algebraic expression f_q, which is a function whose input is a collection with schema $\overline{V}$. The algebraic expressions outputs a collection in the *Unified Data Model*. We do not discuss the trivial operators used to convert the collection into an XML-formatted document.

Selection The selection operator σ is defined in the usual way based on a logical formula (predicate) that can be evaluated over all its input tuples. The selection operator outputs those tuples for which the predicate evaluated to true.

Projection The projection operator π is defined by specifying a list of column names to be retained in the output. The operator π does not eliminate duplicates; we denote duplicate-eliminating projections by π^0.

Navigation The navigation operator follows the established tree pattern paradigm [[reference to XML Tree Pattern entry]]. XQuery tuple-based algebras involve tree patterns where the root of the navigation may be a variable binding, in order to capture nested XQueries where a navigation in the inner query may start from a variable of the outer query. Furthermore, XQuery tuple-based algebras have extended tree patterns with an option for "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 is similar to 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. In what follows, the tree patterns are limited to child and descendant navigation. The literature describes extensions to all XPath axes.
An *unordered tree pattern* is a labelled tree, whose nodes are labelled with (*i*) an 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. The variables appearing in the tree pattern form the *schema* of the tree pattern.
The semantics of the navigation operator is based the *mapping* of a tree pattern to a value. Given a variable-value pair $\$R = r_i$ and a tree pattern T with schema V, a mapping of T to r_i is a mapping of the pattern nodes to nodes in the XML tree represented by r_i such that the structural relationships among pattern nodes are satisfied.

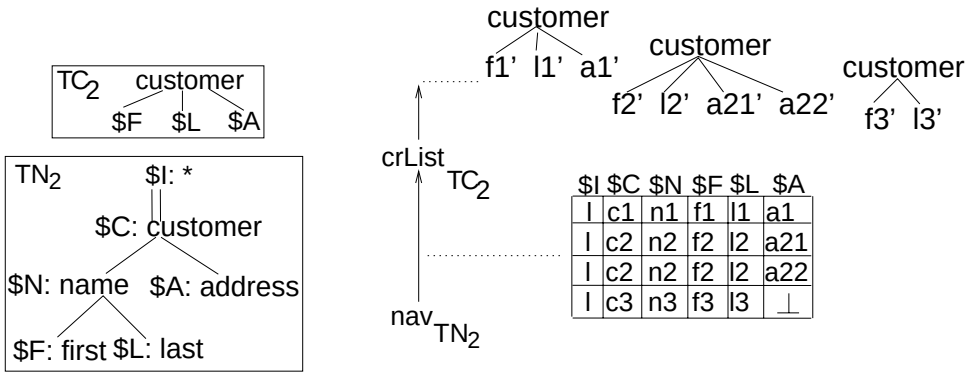

Figure 3: Navigation and construction for Q2.

A *mapping tuple* (also called *binding*) has schema V and pairs each variable in V to the node to which it is mapped corresponds to such a mapping. The function $map(T, r_i)$ returns the set of bindings corresponding to all mappings. For a more detailed explanation of a mapping of a tree pattern to a value of the XPath/XQuery data model see [[encyclopedia article on XML Tree Pattern]].
A mapping may be partial: nodes connected to the pattern by optional edges may not be mapped, in which case this node is mapped to a $\perp$.
To formally define embeddings for tree patterns T that have optional edges, we introduce the auxiliary *pad* and *sp* functions.
Given a set of variables $\overline{V}$, the function $pad_{\overline{V}}(t)$ extends the tuple t to have a variable-value pair $\$V = \perp$ for every variable $\$V$ of $\overline{V}$ that is not included in t. i.e., *pad* pads t with nulls so that it has the desired schema $\overline{V}$. We overload the function to apply on a set of tuples S, so that $pad_{\overline{V}}(S)$ extends each tuple of S. Given two tuples t and t', we say that t is *more specific* than t' if for every attribute/variable $\$V$, either $t.\$V = t'.\$V$ or $t'.\$V = \perp$. For example, $[\$A = n_a, \$B = \perp, \$C = \perp]$ is less specific than $[\$A = n_a, \$B = n_b, \$C = \perp]$. Given a set S of tuples we name $sp(S)$ the set that consists of all tuples S that are not less specific than any other tuple of S. For example, $sp(\{[\$A = n_a, \$B = \perp, \$C = \perp], [\$A = n_a, \$B = n_b, \$C = \perp]\}) = \{[\$A = n_a, \$B = n_b, \$C = \perp]\}$.
Then given a tree pattern T with set of variables $\overline{V}$ and optional edges, we create the set of tree patterns $T^1, \ldots, T^n$ that have no optional edges and are obtained by non-deterministically replacing each optional edge with a required edge, or removing the edge and the subtree that is adjacent to it. We then define the embeddings of the pattern T as:

$$sp(pad_{\overline{V}}(map(T^1, I) \cup \ldots \cup map(T^n, I)))$$

Mapping an unordered tree pattern to a value produces unordered tuples. In an *ordered tree pattern* the variables are ordered by the preorder traversal sequence of the tree pattern, and this ordering translates into an order of the attribute values of each result tuple. Tuples in the mapping result are then ordered lexicographically according to the order of their attribute values.
The navigation operator nav_T inputs and outputs a list of tuples with schema V. The parameter T is a tree pattern, whose root must be labeled with a variable $\$R$ that also appears in V. The input to the operator is a list of the form $\langle t_1, \ldots, t_n \rangle$, where each t_i, $i = 1, \ldots, n$ is a tuple of the form $[\ldots, \$R = r_i, \ldots]$. The output of the operator is the list of tuples $t_i + t'_i$, for all i, where t'_i is defined as $t'_i \in map(T, r_i)$.
Figure 2 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 $a_{21} << a_{22}$.

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 collection of tuples and outputs an XML tree/forest. The parameter L is a list of *construction tree patterns*, called a *construction pattern list*. For example, consider the query:

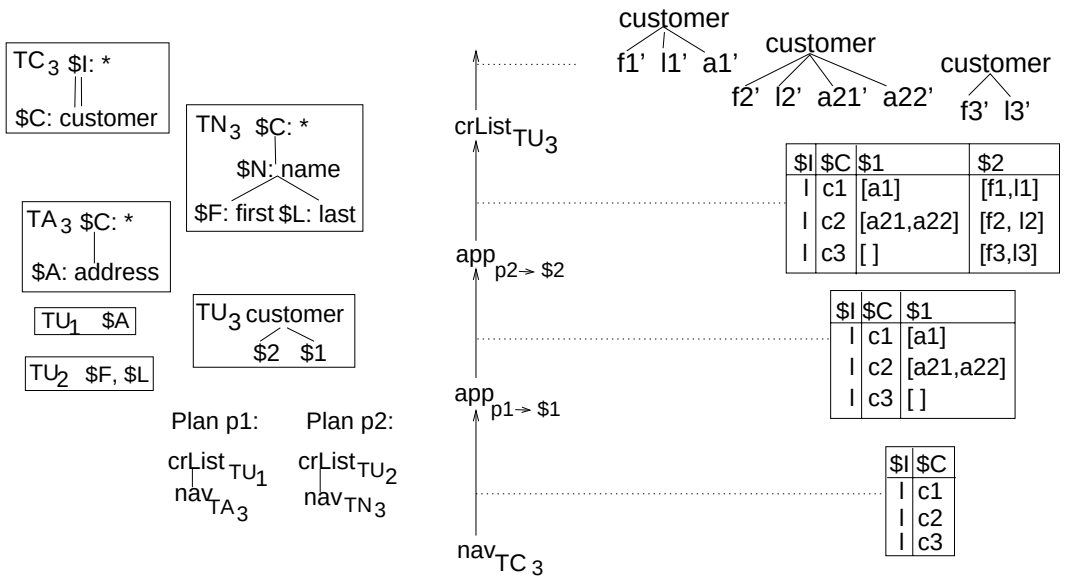

Figure 4: Algebraic plan for Q1.

```
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 3 depicts the navigation pattern TN_2, the construction pattern list TC_2, which consists of a single construction pattern, and a simple algebraic expression, corresponding to Q2.

Nested Plans The combination of navigation and construction operators captures the navigation and construction of unnested FLWR XQuery expressions. The following operators can be used to handle nested queries.

Apply The $app_{p \mapsto \$R}$ (as in "*app*ly plan") unary operator takes as parameter an algebra expression p, which delivers an XML "forest", and a result variable $\$R$, which should not appear in the schema V of the input tuples. Intuitively, for every tuple t in the input collection I, $p(\{t\})$ is evaluated and the result is assigned to $\$R$.

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

For example, query Q1 produces 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 4 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 construction pattern lists, 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 and, respectively, \$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 $app_{p1 \mapsto \$1}$ on one tuple at a time (which means twice for customer $c2$ since she 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 $\overline{G_v}$

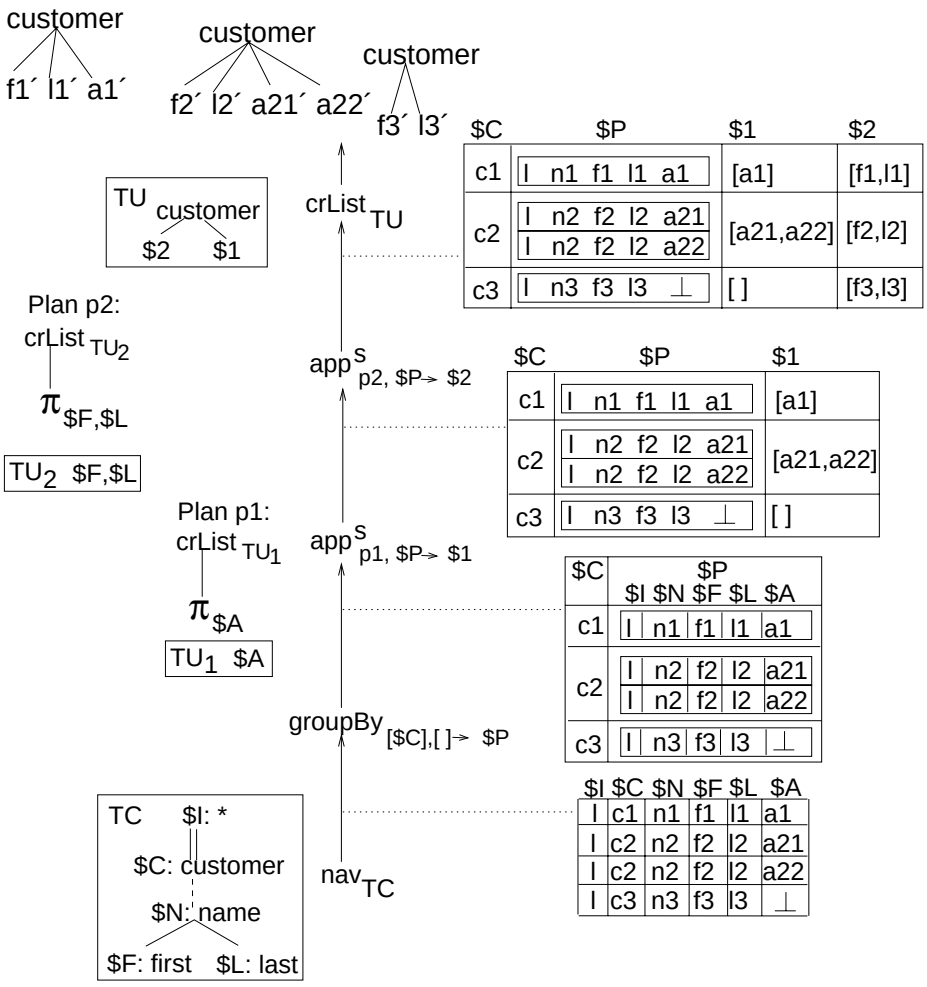

Figure 5: Alternative algebraic plan for Q1.

and an additional variable $\$R$, whose value is the partition.[2] Note that we do not consider input tuples that have 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 collection 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 $\$R$.

For example, Figure 5 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.

Comparing Figure 5 with Figure 4 shows that the same query may be expressed by different expressions in the unified algebra. Providing multiple operators (or combinations thereof) which lead to the same compuation is typical in algebras.

KEY APPLICATIONS

An XML tuple algebra is an important intermediate representation for the investigation and the implementation of efficient XML processing techniques. An XML tuple algebra or similar extensions to relational algebra are used

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

as of 2008 by XQuery processing systems such as BEA Aqualogic Data Services Platform and the open source MonetDB/XQuery system.

FUTURE DIRECTIONS

Optional.

EXPERIMENTAL RESULTS

Optional.

DATA SETS

Optional.

URL TO CODE

Optional.

CROSS REFERENCE

XML document
XML element
XML query processing
XML storage
XML Tree Pattern/XML Twig Query
XPath/XQuery

RECOMMENDED READING

Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

[1] Andrei Arion, Véronique Benzaken, Ioana Manolescu, Yannis Papakonstantinou, and Ravi Vijay. Algebra-based identification of tree patterns in XQuery. In *FQAS*, pages 13–25, 2006.
[2] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB*, 1999.
[3] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. Technical report, 1995.
[4] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT logical framework for XQuery. In *VLDB*, 2004.
[5] Philippe Michiels, George A. Mihaila, and Jérôme Siméon. Put a Tree Pattern in Your Algebra. In *ICDE*, 2007.
[6] Yannis Papakonstantinou, Vinayak R. Borkar, Maxim Orgiyan, Konstantinos Stathatos, Lucian Suta, Vasilis Vassalos, and Pavel Velikhov. XML queries and algebra in the Enosys integration platform. *Data Knowl. Eng.*, 44(3):299–322, 2003.
[7] Christopher Re, Jérôme Siméon, and Mary Fernández. A complete and efficient algebraic compiler for XQuery. In *ICDE*, 2006.
[8] XQuery 1.0 and XPath 2.0 Data Model. www.w3.org/TR/xpath-datamodel.
[9] The XQuery language. www.w3.org/TR/xquery, 2004.
[10] XQuery 1.0 Formal Semantics. www.w3.org/TR/2005/WD-xquery-semantics.

TITLE

SYNONYMS

DEFINITION

MAIN TEXT

CROSS REFERENCE

REFERENCES