Yet Another Query Algebra For XML Data*

Carlo Sartiani Antonio Albano
Dipartimento di Informatica - Università di Pisa
Corso Italia 40, Pisa, Italy
{sartiani,albano}@di.unipi.it

Abstract

XML has reached a widespread diffusion as a language for representing nearly any kind of data source, from relational databases to digital movies. Due to the growing interest toward XML, many tools for storing, processing, and querying XML data have appeared in the last two years.

Three main problems affect XML query processing: path expression evaluation, nested query resolution, and preservation of document order. These issues, which are related to the hierarchical structure of XML and to the features of current XML query languages, require compile-time as well as run-time solutions.

This paper describes a query algebra for XML data. The main purpose of this algebra, which forms the basis for the Xtasy database management system, is to combine good optimization properties with a good expressive power that allows it to model significant fragments of current XML query languages; in particular, explicit support is given to efficient path expression evaluation, nested query resolution, and order preservation.

1. Introduction

XML has reached a widespread diffusion as a language for representing nearly any kind of data source, from relational databases to digital movies. While the usual application for XML is data exchange, there exist many application fields where direct manipulation of XML is needed, e.g., the management of medical data. Therefore, many tools for storing, processing, and querying XML data have appeared in the last two years: some of these tools are based on existing database management systems [18] extended with the ability to store XML data into relational tables and to process XML queries, while others [17, 1] are designed from scratch for XML data. Systems in the first class are designed and implemented by middleware layers that store XML documents into relational tables or collections of objects, and that directly map XPath [8] and XQuery [5] queries into special-purpose SQL queries, endowed with some code needed to fill the expressive gap between XQuery and SQL (even SQL3). Systems in the second class, instead, represent an attempt to build XML-tailored database systems, i.e., systems designed to manage ordered tree-structured data; nevertheless, most of these systems are just based on existing relational, object-oriented, or even *hierarchical* database engines on top of which XML functionalities are built.

Whatever class of systems you are considering, three key problems affect query processing over XML data: *evaluation of path expressions*; *evaluation and resolution of nested queries*; *order preservation*. These problems are related both to the tree structure of XML and to the features offered by current XML query languages.

Path expressions Path expression evaluation requires to traverse a tree according to a given path specification. This specification usually gives only a partial description of the path, by using *wildcards* and recursive operators (e.g., $*$ in GPE [21] and // in XPath). There exist many approaches to path expression optimization. The most popular (and maybe the most effective) ones are based on path indexes or full-text indexes [14, 6], and on path expression minimization [11]. The former approach is based on the massive use of path indexes or full-text indexes, hence trying to solve this optimization problem at the physical level only; the latter approach, instead, is based on the fusion of the path expression automaton with schema information. Both kinds of approaches exploit structural information about XML data (usually DTD-like schemas).

Another interesting approach tries to expand path expressions at compile-time [19], by replacing recursive operators with real paths being present in the data; substitution information is taken from a *DataGuide* [15], a graph containing each path being present in the database. As a matter of fact, this technique can be considered as a special case of

*Research partially supported by the MURST DataX Project and by Microsoft Research

path expression minimization.

Nested queries Current XML query languages impose no restriction on query nesting: indeed, they generalize the "free nesting philosophy" of OQL, and allow one to put a query or a complex expression returning a *well-formed* document wherever a well-formed XML document is expected. This feature allows one to easily formulate complex queries, e.g., queries containing esoteric joins, or queries changing the structure of data (see [5] and [12] for a discussion on how to use nested queries for grouping and reshaping XML elements). As a consequence, nested query resolution has become more and more important, at least to transform annoying d-joins into more tractable ordinary joins.

Ordering Unlike relational data models, XML is an **ordered** data format, i.e., a total order is defined among elements of the same document, as well as a total order among the children of a given node. A common requirement in XML applications, such as the managing of digital movies, is the preservation of document order among elements extracted from the database; still, there are many application fields (e.g., database publishing and semistructured database management) that do not require to retain document order. Most XML query languages support these requirements, and also allow the user to specify an arbitrary order among elements in query results (the way document order and user-defined order combine together is unclear).

The order preservation problem also has another face. Some query languages, such as XQuery, express tree navigation by means of d-joins, which are inherently ordered, e.g., $A < B > \neq B < A >$. Unfortunately, d-joins are also used for connecting independent path expressions. XQuery formal semantics imposes that d-joins are evaluated in the query order, until an explicit statement by the user is given. As a result, order preservation in XML queries requires one to deal with these three issues: document order, user-defined order, and join order preservation.

Our contribution This paper shows a query algebra for XML data. This algebra, which forms the basis for the Xtasy database management system [10], has been defined as an extension of object-oriented and semistructured query algebras [9, 7, 4]; it retains common relational and OO optimization properties (e.g., join commutativity and associativity), and gives explicit support to efficient path expression evaluation, nested query resolution, and order preservation. In particular, the algebra provides general rewriting rules for transforming d-joins into ordinary joins, as well as a general approach for preserving order in XQuery queries.

The paper is structured as follows. Section 2 describes the Xtasy data model; next, Section 3 describes the algebra operators. Then, Sections 4 introduces some algebraic equivalences. Next, Section 5 contains a review of related works. Finally, in Section 6 we draw our conclusions.

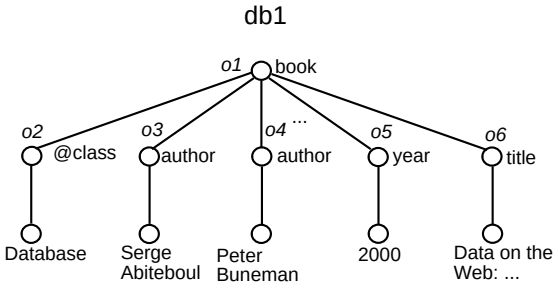

Figure 1. A data model instance

2. Data model and term language

The Xtasy query algebra employs a data model similar to the W3C XML Query Data Model [13]. A data model instance is a *well-formed* XML document represented as an unordered forest of *node-labeled* trees, the global ordering being preserved by a special-purpose function pos; internal nodes are labeled with constants (tags and attribute names), and leaves with atomic values. Each internal node has a unique *object identifier* (oid) that can be accessed by the special-purpose function oid; an algebraic support operator ν is used to generate new oids and to refresh existing ones, hence allowing the algebra to support *copy* semantics as well as *reference* semantics operations.

Example 2.1 Consider the XML fragment shown below:

```
<book class = "Database">
   <author> Serge Abiteboul </author>
   <author> Peter Buneman </author>
   <author> Dan Suciu </author>
   <title> Data on the web: from ... </title>
   <year> 2000 </year>
   <publisher> ... </publisher>
</book>
```

This fragment can be represented by the tree depicted in Figure 1.

■

3. Algebra operators

Xtasy algebra is an extension of common object-oriented and semistructured query algebras to XML. The starting point of the algebra is the object-oriented algebra described in [9]; from that the Xtasy algebra borrows the idea of relational-like intermediate structures, hence extending to XML common relational and OO optimization strategies, as well as the presence of *border* operators, which insulate

other algebraic operators from the technicalities of XML. The algebra provides two border operators, namely $path$ and $return$, which respectively build up intermediate structures from XML documents and publish these structures into XML.

In addition to $path$ and $return$, the Xtasy algebra provides quite common operators such as $Selection$, $Projection$, $TupJoin$, $Join$, $DJoin$, Map, $Sort$, $TupSort$, and $GroupBy$.

There exist both *set-based* and *list-based* versions of the algebraic operators: list-based operators should ease the management of the forthcoming XPath 2.0 path language [20]. For the sake of brevity, in the following sections only set-based algebraic operators will be presented.

3.1. Env structures

As already stated, algebraic operators manipulate relational-like structures. These structures, called Env, are very similar to YAT Tab structures [7], and contain the variable bindings collected during query evaluation. As in [9] and YAT, Env structures allow one to define algebraic operators that manipulate sets of tuples, instead of trees; hence, common optimization and execution strategies (which are based on tuples rather than trees) can be easily adapted to XML without the need to redefine all that stuff.

An Env structure is a collection of *flat* tuples, each tuple describing a set of variable bindings. With the only exception of sorting operators, each algebraic operator manipulates unordered Env structures, e.g., tuples order is irrelevant.

The following example shows a sample Env structure.

Example 3.1 Consider the Env structure shown below:

$\$b : o_1$	$\$a : o_3$
$\$b : o_1$	$\$a : o_4$
$\$b : o_1$	$\$a : o_5$
...	...

This structure is a set of tuples, where each tuple consists of two fields. The first field ($\$b : \ldots$) contains the oid of a book element, while the second contains the oid of a related author element. Since the book o_1 has three authors, the Env structure contains three tuples referring to o_1. ■

In order to ensure the closure of the algebra, intermediate structures are themselves represented as node-labeled trees conforming to the algebra data model; this kind of representation also allows one to apply useful optimization properties to border operators. Hence, Env structures can be represented as terms like this:

$$\begin{aligned} env[\quad & tuple[label_1[t_{11}], \ldots, label_n[t_{1n}]], \\ & \ldots, \\ & tuple[label_1[t_{k1}], \ldots, label_n[t_{kn}]]] \end{aligned}$$

Each tuple element describes a binding tuple, where $label_i$ are variable names and t_{ji} the corresponding values.

3.2. $path$ and $return$

$path$ The main task of the $path$ operator is to extract information from the database, and to build variable bindings. The way information is extracted is described by an *input filter*; a filter is a tree, describing the paths to follow into the database (and the way to traverse these paths), the variables to bind and the binding style, as well as the way to combine results coming from different paths. Input filters are described by the following grammar:

$$\begin{array}{lll} F & ::= F_1, \ldots, F_n & \text{conjunctive filters} \\ & \mid F_1 \vee \ldots \vee F_n & \text{disjunctive filters} \\ & \mid (op, var, binder)label[F] & \text{simple input filter} \\ & \mid \emptyset & \text{empty filter} \end{array}$$

where $op \in \{/, //, \_\}$
$var \in String \cup \{\_\}$
$binder \in \{\_, in, =\}$

A simple filter $(op, var, binder)label[F]$ tells the $path$ operator a) to traverse the current context by using the navigational operator op, b) to select those elements or attributes having label $label$, c) to perform the binding expressed by var and $binder$, and d) to continue the evaluation by using the nested filter F.

The $path$ operator takes as input a single data model instance and an input filter, and it returns an Env structure containing the variable bindings described in the filter. The following example shows a simple input filter and its application to a sample document.

Example 3.2 Consider the following fragment of XQuery query:

```
FOR $b in book,
    $a in $b//author,
```

This clause traverses the path book//author into the sample document, binding each book and author element to $\$b$ and $\$a$, respectively. This clause can be translated into the following $path$ operation:

$path_{(\_,\$b,in)book[(//,\$a,in)author[\emptyset]]}(db1)$

which returns the following Env structure:

$\$b : o_1$	$\$a : o_3$
$\$b : o_1$	$\$a : o_4$
$\$b : o_1$	$\$a : o_5$

■

Input filters provide a simple but rich path language, containing common path operators, such as / and // (/l

takes as input a collection of nodes, and returns the children of these nodes labeled by l, while $//l$ returns the descendants of these nodes labeled by l). No direct support, instead, is given to the resolution of ID/IDREF attributes, e.g., `a/b/@c=>/d` is represented by using joins. Moreover, input filters provide two binding styles (in and $=$), which directly correspond to XQuery binders.

The following example shows the grouping binder $=$.

Example 3.3 Consider the following XQuery clause:

```
FOR $b in book,
LET $a_list := $b//author,
```

This clause traverses the path book//author; each book element is bound to $\$b$, and, for each book element, the whole set of author sub-elements is bound to $\$a\_list$. This clause can be expressed by using the following $path$ operation:

$path_{(\_,\$b,in)book[(//,\$a\_list,=)author[\emptyset]]}(db1)$

which returns the following Env structure:

$\$b : o_1$	$\$a : \{o_3, o_4, o_5\}$

■

As shown by the filter grammar, multiple input filters can be combined to form more complex filters. Xtasy algebra allows filters to be combined in a *conjunctive* way, or in a *disjunctive* way. In the first case, the Env structures built by simple filters are joined together, hence imposing a product semantics; in the second case, partial results are combined by using an *outer union* operation. Therefore, disjunctive filters can be used to map XPath union paths into input filters (e.g., book/(author|publisher)), as well as more sophisticated queries; the use of outer union ensures that the resulting Env has a uniform structure, i.e., all binding tuples have the same fields.

The following examples show the use of disjunctive filters.

Example 3.4 Consider the following XQuery clause:

```
FOR $b in book,
    $p in $b/(author|publisher),
```

This clause binds the $\$p$ variable to publishers and authors of each book. It can be expressed by using the following $path$ operation:

$$path_{(\_,\$b,in)book[(/,\$p,in)author[\emptyset]\vee(/,\$p,in)publisher[\emptyset]]}(db1)$$

■

Due to the presence of disjunction, a precedence order among combinators has to be established: we chose to give precedence to disjunction, i.e., $f_1 \vee f_2, f_3 \vee f_4$ is evaluated as $(f_1 \vee f_2), (f_3 \vee f_4)$.

$return$ While the $path$ operator extracts information from existing XML documents, the $return$ operator uses the variable bindings of an Env to produce new XML documents. $return$ takes as input an Env structure and an *output filter*, i.e., a skeleton of the XML document being produced, and returns a data model instance (i.e., a well-formed XML document) conforming to the filter. This instance is built up by filling the XML skeleton with variable values taken from the Env structure: this substitution is performed once per each tuple contained in the Env, hence producing one skeleton instance per tuple.

Output filters satisfy the following grammar:

$$\begin{aligned} OF &::= OF_1, \ldots, OF_n \\ &\mid label[OF] \\ &\mid @label[val] \\ &\mid val \\ val &::= v_B \mid var \mid \nu var \end{aligned}$$

An output filter may be an *element constructor* ($label[OF]$), which produces an element tagged $label$ and whose content is given by OF, an *attribute constructor* ($@label[val]$), which builds an attribute containing the value val, or a combination of output filters ($OF_1, \ldots, OF_n$). The second production needs further comments. The algebra offers two way to publish information contained in an Env structure: by copy (νvar) and by reference (var). Referenced elements are published as they are in query results; in particular, their object ids are not changed, thus allowing support for the definition and management of views over the database. Copied elements, instead, are published with *fresh* oids, hence losing the ties with their originating databases.

The following example shows the use of the $return$ operator.

Example 3.5 Consider the following XQuery query:

```
FOR $b in book
    $t in $b/title,
    $a in $b/author
RETURN
    <entry> $t, $a </entry>
```

This query returns the title and the authors of each book. This query can be represented by the following algebraic expression:

$$return_{entry[\nu\$t,\nu\$a]}(path_{(\_,\$b,in)book[(/,\$a,in)author[\emptyset],(/,\$t,in)title[\emptyset]]}(db1))$$

■

The following example shows the use of the $return$ operator to define a view over the database.

Example 3.6 Assume that you want to define a database view restricting the access to only those books published before 2001. By using a reference output filter this task can be accomplished by the following algebraic expression:

$$return_{view[\$b]}(\sigma_{\$y<2001}(path_{(\_,\$b,in)book[(/,\$y,in)year[\emptyset]]}(db1)))$$

■

3.3. Basic operators

Xtasy algebra basic operators manipulate Env structures only, and perform quite common operations. They resemble very closely their relational or object-oriented counterparts, thus allowing the query optimizer to employ usual algebraic optimization strategies. This class contains Map, $TupJoin$, $Join$, $DJoin$, $Selection$, $Projection$, $GroupBy$, $Sort$, as well as $Union$, $Intersection$, $Difference$, $OuterUnion$, and their list-based counterparts. In the following the most important operators will be presented.

Selection $Selection$ σ takes as input an Env and a boolean predicate P, and returns a new env structure where binding tuples not satisfying P are missing. The predicate language of the Xtasy algebra is quite rich, offering existential as well as universal quantification over variables. These quantifications are required for easily translating universally quantified XQuery queries, and can be optimized by using quite standard rewriting techniques.

TupJoin $TupJoin$ $\bowtie_P$ is the Xtasy counterpart of standard join operators. So, it takes as input two Env structures e_1 and e_2 as well as a boolean predicate P; it evaluates the predicate P over each pair of tuples $(t_1, t_2) \in e_1 \times e_2$, returning only the pairs satisfying P. The primary use of the $TupJoin$ operator is to combine $path$ operations over independent data sources, and it is also introduced during query unnesting. The following example shows the typical use of $TupJoin$.

Example 3.7 Consider the following query fragment:

```
FOR $b in book,
    $a in $b/author,
    $b1 in document("amazoncatalog.xml")/book,
    $t in $b1/title
```

This query accesses two data sources, an internal one ($db1$) and an external one (`catalog.xml`). This fragment can be represented by the algebraic expression shown below.

$$(path_{(\_,\$b,in)book[(/,\$a,in)author[\emptyset]]}(db1)) \bowtie_{true} (path_{(\_,\$b1,in)book[(/,\$t,in)title[\emptyset]]}(extdb))$$

■

The previous example requires further comments. Unlike XQuery joins, Xtasy algebra joins are unordered, e.g., $e_1 \bowtie_P e_2 \equiv e_2 \bowtie_P e_1$. This is a significant divergence from XQuery Formal Semantics, since XQuery Formal Semantics joins are (unless otherwise stated by the user) *non-commutative*, even on independent operands, i.e., $e_1 \bowtie_P e_2 \neq e_2 \bowtie_P e_1$. This divergence imposes the use of a $Sort$ operation before the $return$ operation, as it will be shown in the next paragraphs; we chose this approach since we believe that the join efficiency improvement might overcompensate for the cost of the additional $Sort$ operation.

DJoin Unlike the $TupJoin$ operator, which joins together two independent Env structures, the $DJoin <\cdot>$ performs a join between two Env e_1 and e_2, where the evaluation of e_2 may depend on e_1. This operator comes from object-oriented query algebras, and it is used to translate for and let clauses of XQuery and, in particular, to combine an inner nested block with the outer one.

The only way to evaluate a $DJoin$ is to perform a nested loop among operands, hence one major goal of the optimization process is to transform, whenever possible, $DJoins$ into more tractable $TupJoin$ operations.

Sort The $Sort$ operator is used for dealing with the three sorting issues described in the Introduction: translating the sortby clause of XQuery (and similar clauses of other languages), preserving document order, and retaining join order. $Sort$ takes as input an Env structure e and an ordering predicate P, and returns e sorted according to P. Ordering predicates are binary predicates defined on binding tuples, and used to impose the desired order; they have signature: $tuple \times tuple \rightarrow boolean$. The following example shows the use of $Sort$ for translating sortby clauses.

Example 3.8 Consider the following query:

```
UNORDERED(
   FOR $b in book
   RETURN $b
   SORTBY (title))
```

This query just returns the list of all books sorted by title. To translate this query, we need to define an appropriate predicate, as the following: $Pred(u, v) \equiv u.\$t < v.\$t$, where u and v ranges over Env tuples, and $\$t$ is bound to book titles. Thus, this query can be represented by the following algebraic expression:

$$return_{\nu\$b}(Sort_{u.\$t<v.\$t}(path_{(\_,\$b,in)book[(/,\$t,in)title[\emptyset]]}(db1)))$$

■

For preserving join order and order among elements a specialized version of $Sort$ is used (called $TupSort$). $TupSort$ takes as input an ordered list of variables $(\$x_1, \ldots, \$x_n)$, and an Env e; it returns a new Env e' obtained by sorting e according to the position of values of the variables $\$x_1, \ldots, \$x_n$. Hence, the algebra can mimic the behavior of XQuery joins, whose semantics requires to retain the order in which variables are bound, unless the programmer qualifies the query with the keyword UNORDERED. The following example shows how $TupSort$ can be used to preserve order among variables and XML elements.

Example 3.9 Consider the following query:

```
FOR $b in book,
    $t in $b/title,
    $a in $b/author
RETURN  <entry> $t, $a </entry>
```

XQuery semantics [12] prescribes that joins should be executed in an ordered fashion. Hence, a correct translation of this query should contain the $TupSort$ operation $TupSort_{(\$b,\$t,\$a)}(\ldots)$, which sorts tuples in the Env structure according to the order specified in the query. ■

n-ary sortby clauses can be translated by using a n-ary ordering predicate, or by a combination of unary $Sort$ operations. The following example shows the translation of such sortby clauses.

Example 3.10 Consider the query of Example 3.8 and assume that we want to return books ordered by title and by author.

```
UNORDERED(
   FOR $b in book
   RETURN $b
   SORTBY (title, author))
```

Unlike [12], the Xtasy algebra offers two ways to translate this query. The first augments the ordering predicate of the $Sort$ operation: $(u.\$t < v.\$t) \vee (u.\$t = v.\$t \wedge u.\$a = v.\$a)$ where $a is bound to each book author; the second one breaks the sortby clause into two smaller clauses, as shown below:

$$return_{\nu\$b}(Sort_{u.\$t<v.\$t}(Sort_{u.\$a<v.\$a}(path_{(\_,\$b,in)book[(/,\$t,in)title[\emptyset],(/,\$a,in)author[\emptyset]]}(db1))))$$

■

GroupBy The $GroupBy$ operator Γ of Xtasy takes as input an Env structure e, and partitions it according to the following definition: $\Gamma_{g;A;f_1;f;\theta}(e) = \{y.A\bullet[g:G] \mid y \in e, G = f(\{x \mid x \in e, f_1(x)\theta f_1(y)\})\}$ where $A \subseteq Att(e)$ and $y \notin Att(e)$.

As shown by the definition (very close to that of [9]), Xtasy $GroupBy$ projects e tuples over A, and augments them with the corresponding groups G, obtained by applying the function f to the set of related tuples.

4. Optimization properties

Three classes of algebraic equivalences can be applied to the Xtasy query algebra. The first class contains *classical* equivalences inherited from relational and OO algebras (e.g., *push-down* of $Selection$ operations and commutativity of joins); the second class consists of path decomposition rules, which allows the query optimizer to break complex input filters into simpler ones; the third class, finally, contains equivalences used for unnesting nested queries. In the next sections, the following notation will be used: (i) $Att(e)$ is the set of labels of an Env structure e; (ii) $FV(exp)$ is the set of free variables occurring in an algebraic expression exp; (iii) $symbols(of)$ is the set of node labels used in the output filter of.

4.1. Classical equivalences

Given the close resemblance of Xtasy algebraic operators to relational and OO operators, the Xtasy algebra supports a wide range of classical equivalences. In particular, $Selection$, $Projection$, Map, $TupJoin$, and even $return$ are linear, so common *reordinability* laws can be easily applied to these operators.

Here follows a brief (and quite incomplete) list of supported algebraic equivalences.

$$\sigma_{P_1 \wedge P_2}(e) \equiv \sigma_{P_1}(\sigma_{P_2}(e)))$$
$$\sigma_{P_1}(\sigma_{P_2}(e)) \equiv \sigma_{P_2}(\sigma_{P_1}(e))$$
$$\sigma_{P_1}((e_1) \bowtie_{P_2} (e_2)) \equiv (e_1) \bowtie_{P_1 \wedge P_2} (e_2)$$
$$(e_1) \bowtie_{Pred} (e_2) \equiv (e_2) \bowtie_{Pred} (e_1)$$

4.2. $path$ decompositions

As already stated, $path$ is the most important operator in the algebra, since it performs the basic tasks of evaluating path expressions and binding variables (both in an iterative fashion and in a grouping fashion). As a result, its efficiency affects the efficiency of the whole query processing. An efficient evaluation of $path$ relies on the ability of the query compiler to simplify path expressions and to exploit existing access support structures, indexes in particular, which can dramatically speed up the evaluation. To this purpose the ability to decompose a complex filter into smaller ones is crucial, since it allows to match existing access structures as well as to replace expensive filters (i.e., filters involving //) with less expensive ones. By using decomposition laws,

optimization techniques based on path indexes and full-text indexes can be safely applied.

The Xtasy algebra provides three decomposition laws for $path$ operations: the first works on the nested structure of a filter, while the remaining ones work on the horizontal structure of a filter.

Proposition 4.1 *Vertical decomposition of $path$ operations*

$$path_{(op,var,binder)label[F]}(t) \equiv path_{(\_,\_,in)env[(/,\_,in)tuple[(/,\_,in)var[(/,var,binder)\_[F]]]]}(path_{(op,var,binder)label[\emptyset]}(t))$$

The following example shows how this decomposition law can be exploited during query optimization.

Example 4.2 Consider the following XQuery clause:

```
FOR $b in library/book,
    $a in $b/author,
    $y in $b/year,
```

This clause retrieves the sub-elements of each book element, binding them to a corresponding variable. This clause can be mapped into the following $path$ operation.

$path_{(\_,\_,in)library[(/,\$b,in)book[F]]}(db_2)$
where
$F \equiv (/,\$a,in)author[\emptyset], (/,\$y,in)year[\emptyset]$

Assume now that a path index on library/book is available. To exploit the presence of this index, the previous $path$ operation should be decomposed into a $path$ operation with filter $(\_,\_,in)library[(/,\$b,in)book[\emptyset]]$ and a new $path$ operation, which further explores the subtrees bound to $\$b$. This decomposition is shown below.

$path_{F_1}(path_{(\_,\_,in)library[(/,\$b,in)book[\emptyset]]}(db2))$
where
$F_1 \equiv (\_,\_,in)env[(/,\_,in)tuple[(/,\_,in)b[F_2]]]$
$F_2 \equiv (/,\$b,in)\_[(/,\$a,in)author[\emptyset], (/,\$y,in)year[\emptyset]]$

■

Proposition 4.3 *Horizontal decomposition of conjunctive input filters*

$$path_{f_1,\ldots,f_m}(t) \equiv (path_{f_1,\ldots,f_{i-1}}(t)) \bowtie_{true} (path_{f_i,\ldots,f_m}(t))$$

Proposition 4.4 *Horizontal decomposition of disjunctive input filters*

$$path_{f_1\vee\ldots\vee f_m}(t) \equiv (path_{f_1\vee\ldots\vee f_{i-1}}(t)) OuterUnion (path_{f_i\vee\ldots\vee f_m}(t))$$

The Xtasy query algebra supports also decompositions of $path$ via d-joins; those decompositions allow the compiler to translate XQuery for and let clause by using arbitrarily complex filters, or simple filters only (as in XQuery Formal Semantics).

Proposition 4.5 *Dependent decomposition of input filters*

$$path_{(op,var,binder)label[F]}(t) \equiv path_{(op,var,binder)label[\emptyset]}(t) < path_F(var) >$$

4.3. Nested queries equivalences

This Section presents some equivalence rules that can be used to transform d-joins induced by nested queries into $TupJoin$ operations. These rules are not intended to be exhaustive, nor to be the most efficient transformations; they just rewrite $DJoins$ induced by nested queries into more tractable joins, and do not exploit *special-purpose* algebraic operators such as binary grouping. Before presenting our rewriting rules a brief introduction to the problem of query unnesting is necessary. In the reference paper about nested queries in object databases [9], authors extend Kim's taxonomy of relational nested queries by defining three classification criteria: the *kind of nesting*, i.e., queries of type A, N, J, JA; the *nesting location*, i.e., the presence of nested queries into the $select$, $from$, or $where$ clause of OQL queries; the *kind of dependency*, i.e., the location of references to external variables. Referring to such classification, our rewriting rules apply to queries of type J (nested dependent queries returning sets of elements/attributes/values), and deal with the three kinds of dependencies (**projection** dependency, **range** dependency, and **predicate** dependency). In the next paragraphs these three kinds of dependencies will be discussed in more detail.

A typical query has the following structure:

$$return_{of}(Sort_{P_1}(\sigma_{P_2}(path_f(db))))$$

The output filter of, the selection predicate P_2 as well as the input filter f can define dependencies with an outer query by referring to external variables. Depending on where these references are located into the nested query, **projection** dependencies (of), **range** dependencies (db), or **predicate** dependencies (P_2) may occur.

Predicate dependency Predicate dependencies occur when an external variable is referenced into the $where$ clause of the inner query, i.e., into the predicate P_2. Consider, for example, the following query returning authors and the list of their papers.

```
FOR $a in library//author
RETURN $a, <publist> FOR $p in library/*,
```

```
                    $aa in $p/author
                    WHERE $aa = $a
                    RETURN $p
</publist>
```

This query contains a nested block (`FOR $p ... RETURN $p`) that scans the papers and returns only those papers written by a given author (`WHERE $aa = $a`). This query can be represented by the following algebraic expression.

$$
\begin{array}{l}
return_{\nu\$a,publist[\nu\$\_var]}(\\
\quad path_{(\_,\_,in)library[(//,\$a,in)author[\emptyset]]}(db) < q >) \\
\text{where} \\
q \equiv path_{(\_,\$\_var,=)\_[\emptyset]}(return_{\$p}(\sigma_{\$aa=\$a}(\\
\quad path_{(\_,\_,in)library[(/,\$p,in)\_[(/,\$aa,in)author[\emptyset]]]}(db))))
\end{array}
$$

In this kind of dependency the predicate P_2 has the form $Pred(\$X, \$Z, \$Y)$, where \$X are external variables, and \$Y and \$Z local variables. In order to remove this dependency (and the related $DJoin$ operation), we need to decompose $Pred(\$X, \$Z, \$Y)$ into $Pred_{Glob}(\$X, \$Z) \wedge Pred(\$Y, \$Z)$, i.e., to separate local variables from global ones, and to transform the $return$ filter.

Proposition 4.6 *Predicate Dependency*

$$
\begin{array}{c}
e_1 < (path_{(\_,\$var,=)}(return_{of}(\\
\sigma_{Pred(\$X,\$Z,\$Y)}(path_{f_1}(db))))) > \\
\\
\equiv \\
\\
\Gamma_{\$var;Att(e_1);\lambda x.(x.\$Z);id;=}(e_1 \bowtie_{Pred_{Glob}(\$X,\$Z)} \\
(path_{f'}(return_{of'}(\sigma_{Pred_{Loc}(\$Y,\$Z)}(path_{f_1}(db))))))
\end{array}
$$

where

- $of' = \_nested[\_result[of], \_env[z_1[\$z_1], \ldots, z_k[\$z_k]]]$
- $f' = (\_, \$\_n, in)\_nested[(/, \$\_res, =)\_result[\emptyset], (/, \_, in)\_env[(/, \_, in)z_1[(/, \$z_1, =)\_[\emptyset], \ldots, (/, \_, in)z_k[(/, \$z_k, =)\_[\emptyset]]]]]$

if $FV(of) \cap Att(e_1) = \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $\$X \subseteq Att(e_1)$, $\$z_1, \ldots, \$z_k \notin Att(e_1)$, $\$Y \notin Att(e_1)$, $z_1, \ldots, z_k \notin symbols(of)$

By applying this transformation the predicate dependency is brought out of the inner query, hence the previous algebraic expression can be rewritten as follows.

$$
\begin{array}{l}
return_{\nu\$a,publist[\nu\$\_var]}(\\
\quad \Gamma_{\$\_var;\$a;\lambda x.(x.\$aa);id;=}(\\
\quad\quad path_{(\_,\_,in)library[(//,\$a,in)author[\emptyset]]}(db) \bowtie_{\$a=\$aa} q')) \\
\text{where} \\
q' \equiv path_{(\_,\$\_n,in)\_nested[F_1,F_2]}(\\
\quad return_{\_nested\_[result[\$p],\_env[aa[\$aa]]]}(\\
\quad\quad path_{(\_,\_,in)library[(/,\$p,in)\_[(/,\$aa,in)author[\emptyset]]]}(db))) \\
\text{and} \\
F_1 \equiv (/, \$_res, =)\_result[\emptyset] \\
F_2 \equiv (/, \_, in)\_env[(/, \_, in)aa[(/, \$aa, =)\_[\emptyset]]]
\end{array}
$$

Range dependency In this form of dependency, the input filter of the inner query is applied to variables coming from the outer query. Consider, for example, the following query associating each paper with the list of its Italian authors.

```
FOR $p in library/*
RETURN <italianrd>
          $p,
          FOR $a in $p/author
          WHERE $a/country/data() = "Italy"
          RETURN $a
       </italianrd>
```

This query contains an inner block (`FOR $a ... RETURN $a`) retrieving, for each given paper $\$p$, the list of Italian authors (if any). This query can be translated as follows.

$$
\begin{array}{l}
return_{italianrd[\$p,\$\_var]}(\\
\quad path_{(\_,\_,in)library[(/,\$p,in)\_[\emptyset]]}(db) < \\
\quad path_{(\_,\$\_var,=)\_[\emptyset]}(return_{\$a}(\sigma_{\$c=\text{"}Italy\text{"}}(\\
\quad\quad path_{(/,\$a,in)author[(/,\$c,=)country[\emptyset]]}(\$p)))) >)
\end{array}
$$

Object-oriented rewriting rules for range dependencies are based on the use of *type extents*; in particular, if the domains of the external variables referenced by the inner block are covered by type extents, their references are replaced by scans over these extents, and results are then combined through object equality predicates. Such transformations cannot be applied to data without extents, therefore we rely on a different rewriting technique, whose main idea is to **copy** the left part of the $DJoin$ into the nested query, hence transforming it into a constant block, and then to combine results by using an equality predicate.

Proposition 4.7 *Range dependency*

$$
\begin{array}{c}
e_1 < (path_{(\_,\$var,=)\_[\emptyset]}(return_{of}(\\
\sigma_P(path_{f_1,\ldots,f_k}(\$x_1, \ldots, \$x_k))))) > \\
\\
\equiv \\
\\
\Gamma_{\$var;Att(e_1);\lambda y.(y.\$X);id;=}(e_1 \bowtie_{\$X=\$X} \\
(path_{f'}(return_{of'}(\sigma_P(path_{f''}(e_1))))))
\end{array}
$$

where

- $f'' \equiv (\_, \$\_tuple, in)tuple[(/, \_, in)x_1[$
  $(/, \$x_1, =)\_[f_1], \ldots, (/, \_, in)x_k[(/, \$x_k, =)\_[f_k]]]]$
- $of' \equiv \_nested[\_result[of], \_env[x_1[\$x_1], \ldots, x_k[\$x_k]]]$
- $f' \equiv (\_, \$\_n, in)\_nested[(/, \$\_res, in)result[\emptyset],$
  $(/, \_, in)\_env[(/, \_, in)x_1[(/, \$x_1, =)\_[\emptyset]], \ldots,$
  $(/, \_, in)x_k[(/, \$x_k, =)\_[\emptyset]]]]$

if $FV(of) \cap Att(e_1) = \emptyset$, $FV(f_1, \ldots, f_k) \cap Att(e_1) = \emptyset$, $FV(P) \cap Att(e_1) = \emptyset$, $FV(db) = \$X \subseteq Att(e_1)$, $x_1, \ldots x_k \notin symbols(of)$

By applying Proposition 4.7, the previous query can be rewritten as follows.

$return_{italianrd[\nu\$p,\nu\$\_var]}(\Gamma_{\$var;\$p;\lambda y.(y.\$p);id;=}($
$path_{(\_,\_,in)library[(/,\$p,in)\_[\emptyset]]}(db) \bowtie_{\$p=\$p} q_1))$
where
$q_1 \equiv path_{(\_,\$\_n,in)\_nested[F_1,F_2]}($
$return_{\_nested[\_result[\$a],\_env[p[\$p]]]}(\sigma_{\$c=\text{"}Italy\text{"}}($
$path_{(/,\$\_tuple,in)tuple[(/,\_,in)p[(/,\$p,=)\_[F_3]]]}($
$\pi_{\$p}(path_{(\_,\_,in)library[(/,\$p,in)\_[\emptyset]]}(db))))))$
and
$F_1 \equiv (/, \$\_res, in)\_result[\emptyset]$
$F_2 \equiv (/, \_, in)\_env[(/, \_, in)p[(/, \$p, =)\_[\emptyset]]]$
$F_3 \equiv (/, \$a, in)author[(/, \$c, =)country[\emptyset]]$

Projection dependency In this form of dependency the output filter of the inner block refers to external variables. As these variables may be deeply nested into complex XML skeleton and mixed with local variables (user abruptness has no limits), the output filter cannot be decomposed into a local part and a global one. A rule to unnest such dependencies is based on the *copy&join* technique used for range dependencies, as well as on the introduction of cross products. Therefore, the unnested expression may be (much) more expensive than the nested one, hence making this transformation not convenient. For the sake of brevity, this (almost useless) unnesting rule is omitted.

5. Related work

Several algebras for semistructured data and XML have been proposed in the past years. Here we briefly review the most important ones.

YAT YAT [7] is an integration system based on a semistructured tree data model. Its query algebra, largely based on [9], manipulates relational-like intermediate structures. The novelty of this approach is represented by two *frontier* operators, $bind$ and $tree$, which are similar to Xtasy $path$ and $return$: $bind$ expresses binding, vertical navigation, horizontal navigation, as well as grouping operations; $tree$, instead, is used to create new trees, and can perform grouping and sorting operations.

As already stated, the Xtasy query algebra derives from [9], so it is not surprising that they share many common features. However, differences are still present, in particular in the frontier operators and in the sorting policy; while Xtasy $path$ and $return$ are simpler than YAT $bind$ and $tree$ (even though the notation is awkward), YAT $bind$ cannot directly evaluate recursive XPath pattern. Moreover, YAT does not deal with the order preservation problem emerged in the XML context, being mainly intended for virtual data integration.

SAL SAL [4] is a query algebra for XML data based on an ordered data model. SAL is quite similar to the YAT query algebra, even though it requires Map operations to perform variable bindings. One key feature of SAL is the ρ operator, which is used to evaluate general path expressions.

TAX TAX [16] is a query algebra developed in the context of the TIMBER project [2]. TAX data model is based on unordered collections of ordered data trees, and each TAX operator takes as input collections of data trees, and produces as output collections of data trees. Unlike YAT and SAL, TAX directly manipulates trees without the need for an explicit intermediate structure. Data extraction and binding are performed by using *pattern trees*: pattern trees, which resemble Xtasy input filters, describe the structure of the desired data, and impose conditions on them.

Even though TAX is very promising, its optimization properties and sorting policy remain unclear; in particular, the way document and join order is preserved is not well defined, and exposes the algebra to possible clashes between unordered collections and ordered trees. Moreover, the way existing optimization properties may be extended to TAX operators is far from being self-evident.

XQuery Formal Semantics (former XML Query Algebra) XML Query Algebra [12] comes from the activity of the W3C XML Query Working Group, being one of the building blocks of the W3C XML Query suite. That algebra is mainly intended for the formal definition of query languages semantics, and its main contribution is an interesting and powerful type system used for statically inferring query result type. The algebra itself is an abstract version of XQuery, where high-level operators (e.g., n-ary for and sortby clauses) are mapped into low-level algebraic operators. Rewriting rules are provided resembling functional programming languages rules and nested relational rules.

While the XML Query algebra would be very useful in defining query language formal semantics and in studying

typing problems on XML queries, it appears unlikely that it will form the basis for effective implementations of XML query languages.

Other algebras In [3] authors present a query algebra for ordered XML data, which are modeled as rooted graphs. The distinctive feature of this algebra, very influential in the XML community, is the use of ordered algebraic operators; in particular, joins are ordered, i.e., $A \bowtie B \neq B \bowtie A$, even when A and B are independent.

6. Conclusions and future work

This paper describes a query algebra for XML data, as well as some basic optimization properties; this algebra is used in the Xtasy database management system, which is currently under development.

Our future work moves along three lines. First, we are currently implementing a persistent version of Xtasy, and we are exploring the dark world of run-time query processing. Second, we need to investigate further the problem of query unnesting: we believe that a classification of nested queries over XML data could be very useful. Finally, we plan to explore further the problem of order preservation and its effects on query execution costs.

7. Acknowledgments

Authors would like to thank Dario Colazzo for his precious suggestions and his continuous support during the writing of the paper.

References

[1] `http://www.tamino.com`.

[2] `http://www.eecs.umich.edu/db/timber/`.

[3] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for xml. Note to the W3C XML Query Working Group, 1999.

[4] C. Beeri and Y. Tzaban. Sal: An algebra for semistructured data and xml. In *Proceedings of the ACM SIGMOD Workshop on The Web and Databases (WebDB'99), June 3-4, 1999*, Philadelphia, Pennsylvania, USA, June 1999.

[5] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, June 2001. W3C Working Draft.

[6] V. Christophides, S. Cluet, G. Moerkotte, and J. Siméon. Optimizing generalized path expressions using full text indexes. *Networking and Information Systems Journal*, 1(2):177–194, 1998.

[7] V. Christophides, S. Cluet, and J. Siméon. Semistructured and Structured Integration Reconciled: YAT += Efficient Query Processing. Technical report, INRIA, Verso database group, November 1998.

[8] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, Nov. 1999. W3C Recommendation.

[9] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical report, University of Karlsruhe, 1994.

[10] D. Colazzo, P. Manghi, and C. Sartiani. Xtasy: A typed xml database management system. Available at `http://www.di.unipi.it/~sartiani/papers/mementomori.pdf`, 2001.

[11] A. Deutsch and V. Tannen. Optimization Properties for Classes of Conjunctive Regular Path Queries. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01), Frascati, Rome, September 8-10, 2001*, 2001.

[12] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. Technical report, World Wide Web Consortium, June 2001. W3C Working Draft.

[13] M. Fernandez and J. Robie. XML Query Data Model. Technical report, World Wide Web Consortium, May 2000. W3C Working Draft.

[14] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proceedings of the third International Workshop WebDB 2000*, pages 125–136, 2000.

[15] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann, 1997.

[16] H. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for xml. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01), Frascati, Rome, September 8-10, 2001*, 2001.

[17] C.-C. Kanne and G. Moerkotte. Efficient storage of xml data. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, 2000.

[18] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with xml and relational. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

[19] J. McHugh and J. Widom. Compile-time path expansion in Lore. Technical report, Stanford University Database Group, November 1998.

[20] S. Muench, M. Scardina, and M. Fernandez. XPath Requirements Version 2.0. Technical report, World Wide Web Consortium, Feb. 2001. W3C Working Draft.

[21] D. Quass, A. Rajaraman, Y. Sagiv, and J. Ullman. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95, Singapore, December 4-7, 1995*, pages 319–344, 1995.