Yet Another Query Algebra For XML Data*
Draft

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

October 1,2001

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 queries 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; a precondition for the development of such solutions is the presence of a well-founded query algebra that allows one to preserve common relational and object-oriented optimization techniques, and that forms an unifying framework for new techniques.

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, which allows it to model significant fragments of current XML query languages; in particular, explicit support is given to efficient path expressions evaluation, nested queries resolution, and order preservation.

1 Introduction

XML has reached a widespread diffusion as language for representing nearly any kind of data source, from relational databases to digital movies (e.g., the upcoming MPEG-7 format [Mar01]). While the usual application for XML is data exchange, there exist many application fields where direct manipulation of XML is needed (e.g., management of medical data [med]). 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 [MFK+00], while others [KM00, Tam] are designed from scratch for XML data.
Three key problems affect query processing over XML data:

1. evaluation of path expressions;

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

2. evaluation and resolution of nested queries;

3. ordering preservation.

These problems are related both to the tree structure of XML and to the features offered by current XML query languages (e.g., XQuery [CCF+01]).

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 [QRSU95] and // in XPath [CD99]). There exist many approaches to path expression optimization. The most popular (and maybe the most effective) ones are the use of path indexes or full-text indexes [FM00] [CCMS98], and path expression minimization [DT01b, DT01a, CGLV01]. 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 exploits structural information about XML data.

Another interesting approach tries to expand path expressions at compile-time [MW98], by replacing recursive operators with real paths being present in the data; substitution information is taken from a *DataGuide* [GW97], 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 path expression minimization.

Nested queries Current XML query languages, in particular Quilt [CRF00] and XQuery [CCF+01], impose no restriction on query nesting: indeed, they generalize the "free nesting philosophy" of OQL [ASL89], 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. As a consequence, nested queries resolution has become more and more important, at least to transform annoying dependency joins into more tractable ordinary joins.

Ordering Unlike relational data models, XML is an **ordered** data format. Many application fields (e.g., database publishing and semistructured database management) do not require the preservation of order among elements, hence query processing goes as usual. Still, there are some significant applications, such the managing of digital movies, which explicitly require the preservation of document ordering. Further, some query languages, such as XQuery, strongly support *ordered* joins, i.e., $A \bowtie B \neq B \bowtie A$, as well as the ability to impose arbitrary odering on query results (e.g., *sort* clause). Hence, ordering preservation in XML queries requires to combine these requirements (sometimes conflicting).

There exist algebras which do not preserve ordering at all, as well as algebras whose operators are inherently ordered, i.e., each algebra operator preserves order among elements [BMR99]. This second approach, while very influential in the document community and in some W3C committee, has one significant drawback: the loss of join *commutativity*, and of the ability to arbitrarily change joins order.

Our contribution This paper shows a query algebra for XML data. This algebra, which forms the basis for the Xtasy database management system [CMS01], has been defined as an

extension of object-oriented and semistructured query algebras [LMS+93, CM93, CCS98, BT99]; it retains common relational and OO optimization properties (e.g., joins commutativity and associativity), and gives explicit support to efficient path expression evaluation, nested queries resolution, and order preservation. In particular, the algebra provides general rewriting rules for transforming dependency 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 and 5 introduce some algebraic equivalences and discuss the expressive power of the proposed algebra. Next, Section 6 contains a review of related works. Finally, in Section 7 we draw our conclusions.

2 Data Model and Term Language

The proposed algebra employs a simplified version of the W3C XML Query Data Model [FR00]. A data model instance is an unordered collection of *persistence* roots; roots can describe both external data sources and local sources, each source being a (possibly virtual) *well-formed* XML document. Therefore, persistence roots can represent concrete XML documents, stored into the local database, as well as XML views over relational databases. A unique identifier is associated to each persistence root.

Each document, in turn, is 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.

Example 2.1 Consider the XML fragment shown below:

```
<book class = "OpSys">
   <author> Stuart Madnick </author>
   <author> John Donovan </author>
   <title> Operating Systems </title>
   <year> 1974 </year>
</book>

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

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

■

XML documents can also be represented as terms conforming to the following grammar (quite close to the term grammar of [HP00]):

$$
\begin{array}{ll}
(1) & t ::= t_1, \ldots, t_n \quad | \quad label[t] \quad | \quad @label[v_B] \quad | \quad v_B \\
(2) & label ::= as \;\; defined \;\; by \;\; XML \;\; specifications \\
(3) & v_B \in Integer \cup String \cup Boolean \cup \ldots
\end{array}
$$

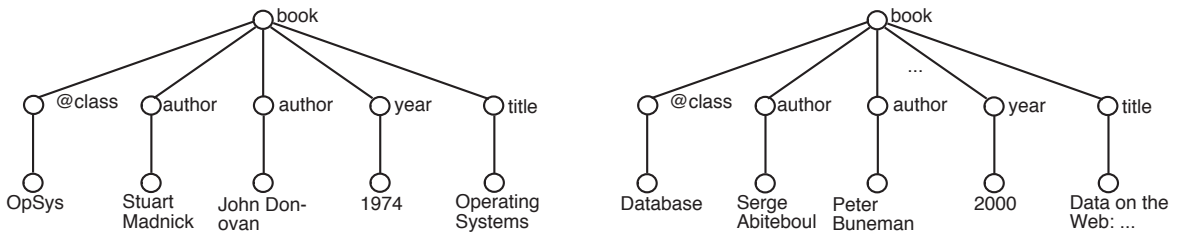

Figure 1: A data model instance

Example 2.2 The fragment shown in the previous example can be described by the following term:

```
book[
   @class["OpSys"],
   author["Stuart Madnick"],
   author["John Donovan"],
   title["Operating Systems"],
   year["1974"]
],
book[
   @class["Database"],
   author["Serge Abiteboul"],
   author["Peter Buneman"],
   auhtor["Dan Suciu"],
   title["Data on the web: from relation to ..."],
   year["2000"],
   publisher["..."]
]
```

■

In the data model only elements and attributes are represented, hence discarding processing instructions, comments, etc. Moreover, no special treatment is given to ID-type and IDREF-type attributes, as well as to linking mechanisms such as XLink.

Two auxiliary functions are defined on XML nodes: *label* and *pos*. $label(t)$ returns the label of the node t, while $pos(t)$ returns an integer denoting the position of t into the global ordering of its surrounding document.

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 YAT query algebra, described in [CCS98] and in more detail in [Sim99]; 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

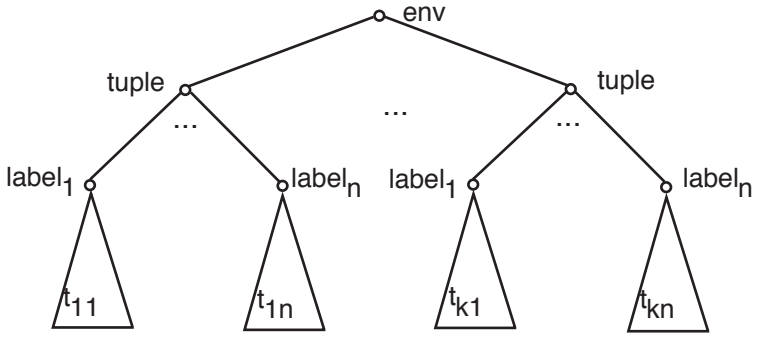

Figure 2: An intermediate structure

return, which respectively build up intermediate structures from XML documents and publish these structures into XML; these operators are quite different from those of YAT, since they allow direct evaluation of recursive XPath patterns, and cannot handle complex grouping and sorting operations as YAT *bind* and *tree*; these operations, instead, are performed by other algebraic operators, namely *GroupBy* and *Sort*.

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. 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 [MSF01]. For the sake of brevity, in the following sections only set-based algebraic operators will be presented.

3.1 Preliminaries

3.1.1 *Env* structure

As already stated, algebraic operators manipulate relational-like structures. These structures, called *Env*, are very similar to YAT *Tab* structures [CDSS98] [Sim99], and contain the variable bindings collected during query evaluation. As in [CM93] 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 an unordered collection of tuples, each tuple describing a set of variable bindings. Since the Xtasy algebra provides both set-oriented and list-oriented operators, there exist two corresponding flavors of *Env* structures; moreover, two operators (*set* and *list*) are provided for converting sets to lists and vice versa.

As shown in Figure 2, *Env* structures are modelled as rooted trees; each tuple element describes a binding tuple, where $label_i$ are variable names and t_{ji} the corresponding values. The *env* structure depicted above can also be represented by the following term:

$env \equiv env[tuple[label_1[t_{11}], \ldots, label_n[t_{1n}]], \ldots, tuple[label_1[t_{k1}], \ldots, label_n[t_{kn}]]]$

In the following sections, set-based *Env* structured will be denoted as[1]:

$e \equiv \{[label_1 : t_{11}, \ldots, label_n : t_{1n}], \ldots, [label_1 : t_{m1}, \ldots, label_n : t_{mn}]\}$

[1]This notation is borrowed from YAT.

3.2 *Border* Operators

Xtasy query algebra provides two *border* operators: *path* and *return*; they are used for insulating other operators, such as *Join* and *DJoin*, from the nested structure of XML, and they play a key role in the whole algebra.

3.2.1 *path*

The main task of the *path* operator is to extract information from persistence roots, 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:

$$
\begin{array}{ll}
(1) & F ::= F_1, \ldots, F_n \quad | \quad F_1 \vee \ldots \vee F_n \quad | \quad (op, var, binder)label[F] \quad | \quad \emptyset \\
(2) & op \in \{/, //, \_, \forall/, \forall//\} \\
(3) & var \in label \cup \{\_\} \\
(4) & binder \in \{\_, in, =\}
\end{array}
$$

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 expessed 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 3.1 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 (also shown in Figure 3):

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

■

Input filters provide a simple but rich path language, containing common path operators, such as / and //, and their universally quantified version (∀/ and ∀//). 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 Quilt and XQuery binders.

The following example shows the grouping binder =.

Example 3.2 Consider the following XQuery clause:

```
for $b in book,
let $a_list := $b//author,
```

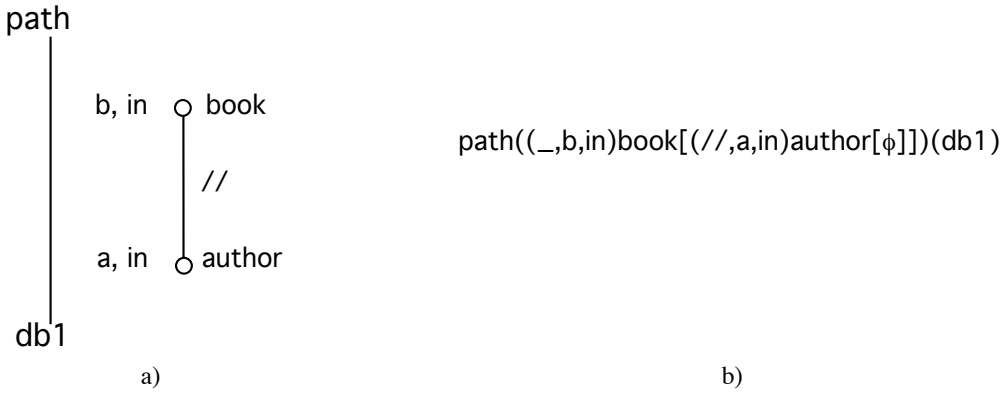

Figure 3: A simple *path* operation

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:

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

■

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 3.3 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; since union paths are not usually used at the beginning of a path, it seems natural to give precedence to disjunction, i.e., $f_1 \vee f_2, f_3 \vee f_4$ is evaluated as $(f_1 \vee f_2), (f_3 \vee f_4)$.

It should be noted that disjunctive filters may be used to express *unsafe* queries, as well as to map queries of potentially unsafe languages (e.g., [CG01]).

As already stated, path expression evaluation is crucial in XML query processing. In order to support the use of various kinds of indexes, decompositions of *path* operations into simpler ones are needed. The Xtasy algebra offers two kind of decompositions: vertical decompositions, which break up a linear path expression into smaller components, and horizontal decompositions,

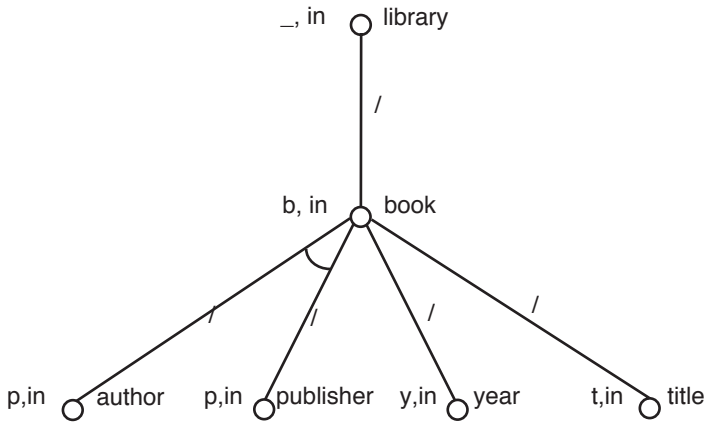

Figure 4: A complex input filter

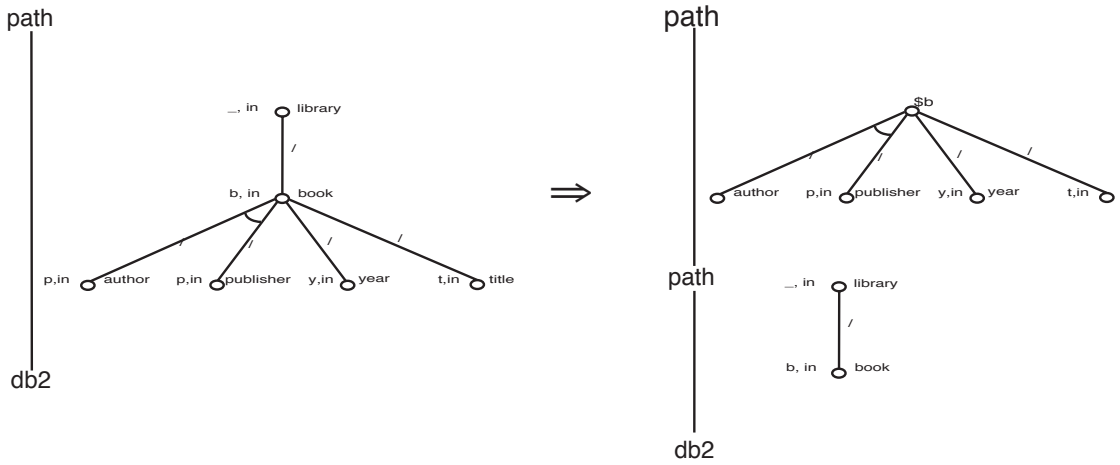

Figure 5: Vertical decomposition of *path*

which break disjunctive or conjunctive filters into their components. These decompositions allow the query optimizer to choose the best evaluation strategy for each filter.

The following example shows how filters can be decomposed. For a complete definition of the decomposition laws see Section 4.

Example 3.4 Consider the following XQuery clause:

```
FOR $b in library/book,
    $p in $b/(author|publisher),
    $t in $b/title,
    $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 a *path* operation using the filter shown in Figure 4.

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 in Figure 5.

■

3.2.2 *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 it 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:

$$(1) OF ::= OF_1, \ldots, OF_n \quad | \quad label[OF] \quad | \quad @label[val] \quad | \quad val$$
$$(2) val ::= v_B \quad | \quad var$$

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 following example shows the use of the *return* operator.

Example 3.5 Consider the following XQuery query:

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

This query returns the title and the authors of each book. This query can be represented by the following algebraic expression (also shown in Figure 6):

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

■

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; this allows 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.

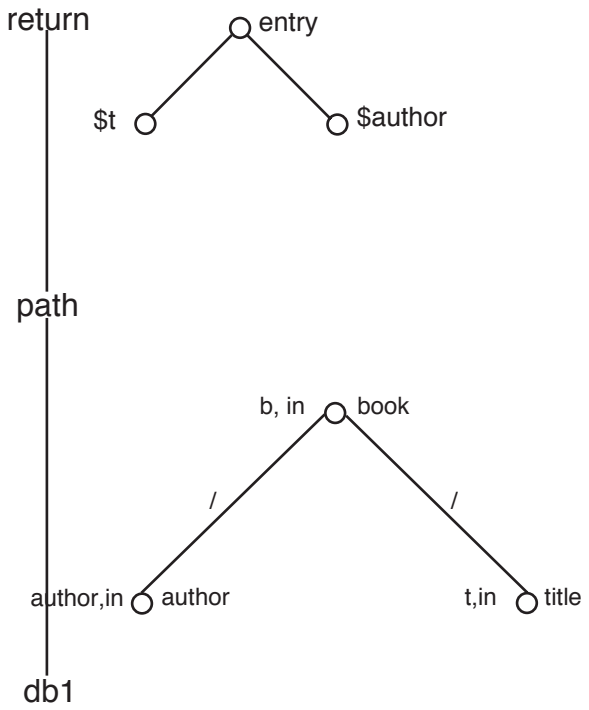

Figure 6: A query containing *return*

Selection *Selection* takes as input an *Env* and a boolean predicate, and it returns a new *env* structure where binding tuples not satisfying the predicate are missing. The predicate language of the Xtasy algebra is quite rich, and it offers existential as well as universal quantification over variables. These quantifications are required for easily translating universally quantified XQuery queries. The following example shows the use of the *Selection* operator.

Example 3.6 Consider the following XQuery query:

```
FOR $b in book,
    $t in $b/title
WHERE EVERY $a in $b/author SATISFIES $a != "Vassilis Christophides"
RETURN <entry> $t </entry>
```

This query returns the title of each book not written by Vassilis Christophides. This query can be represented by the following algebraic expression:

$$return_{entry[\$t]}(\sigma_{\forall \$\alpha \in \$a : \$alpha \neq "VassilisChristophides"} (path_{(\_,\$b,in)book[(/,\$t,in)title[\emptyset],(/,\$a,=)author[\emptyset]]}(db1)))$$

The selection predicate compares the content of each author element with "Vassilis Christophides", and it returns true if and only if each author element content is not equal to "Vassilis Christophides".

■

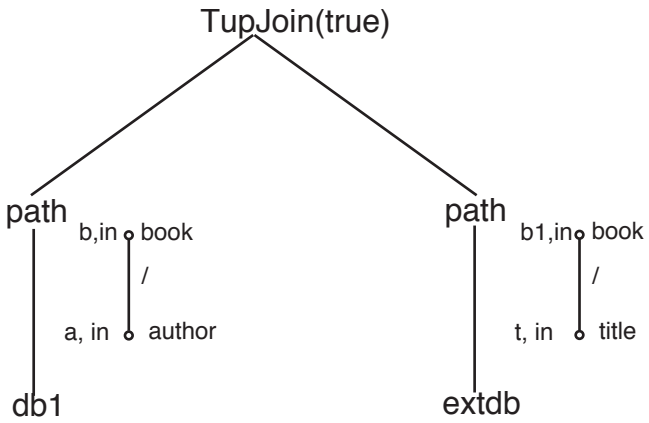

Figure 7: A query containing $TupJoin$

TupJoin $TupJoin$ takes as input two Env structures e_1 and e_2, and a boolean predicate P; it evaluates the predicate P over each pair of tuples $(t_1, t_2) \in e_1 \times e_2$, and it returns an Env containing the concatenation of the tuples satisfying P. $TupJoin$ is primarily used for connecting *path* operations over distinct data sources. The following example shows the 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 (see also Figure 7).

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

■

Unlike XQuery joins, Xtasy algebra joins are **unordered**. This means that $e_1 \bowtie_P e_2 \equiv e_2 \bowtie_P e_1$. Therefore, a correct translation of the previous fragment requires a $Sort$ operation, as it will be shown in the next paragraphs.

Join Unlike $TupJoin$, which has a fixed tuple combination function, the $Join$ operator takes as input a predicate, two Env structures, and a combination function f. f is used to combine tuples in the resulting Env structure. As a consequence, $Join$ is a higher order operator. The $Join$ operator has been inserted to enhance the flexibility of the algebra, and to provide room for future extensions.

DJoin Unlike the $TupJoin$ operator, which joins together two independent Env structures, the $DJoin$ 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 netsed 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 joins.

The following example shows the use of $DJoin$ during query translation.

Example 3.8 Consider the following query:

```
FOR $b in book,
    $t in $b/title
WHERE EVERY $a in $b/author SATISFIES $a != "Vassilis Christophides"
      AND $b/publisher IN Q
RETURN <entry> $t </entry>
```

This query returns the title of each book not written by Vassilis Christophides, whose publisher is contained into the result of a nested query Q. This query can be translated into the following algebraic expression:

$$return_{entry[\$t]}(\sigma_{\forall \alpha \in \$a:\alpha \neq "VassilisChristophides" \wedge \$p \subseteq \$\_var}((path_{(\_,\$b,in)book[(/,\$t,in)title[\emptyset],(/,\$a,=)author[\emptyset],(/,\$p,=)publisher[\emptyset]]}(db1)) < path_{(\_,\$\_var,=)\_[\emptyset]}(Q) >)))$$

■

Sort The $Sort$ operator is used for both translating the $SORTBY$ clause of XQuery (and similar clauses of other languages) and preserving document order. $Sort$ takes as input an Env structure e and an ordering predicate P, and it returns e sorted according to P. Ordering predicates are binary predicates defined on binding tuples, and used to impose the desired order. The following example shows the use of $Sort$ for translating $SORTBY$ clauses.

Example 3.9 Consider the following query:

```
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 $\$t$ is bound to book titles. Thus, this query can be represented by the following algebraic expression:

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

■

For preserving order among elements a specialized version of *Sort* is used (called *TupSort*). *TupSort* takes as input a list of variables ($\$x_1, \ldots, \$x_n$), and an *Env* e; it returns e sorted according to the following ordering predicate:

$$
\begin{aligned}
<_{Tup}(\$x_1, \ldots, \$x_n)(u, v) = \quad & (pos(u.\$x_1) < pos(v.\$x_1)) \vee \\
& (pos(u.\$x_1) = pos(v.\$x_1) \wedge pos(u.\$x_2) < pos(v.\$x_2)) \vee \ldots \vee \\
& (pos(u.\$x_1) = pos(v.\$x_1) \wedge \ldots \wedge pos(u.\$x_{n-1}) = pos(v.\$x_{n-1}) \wedge \\
& pos(u.\$x_n) < pos(v.\$x_n))
\end{aligned}
$$

This predicate allows the algebra to mimic the disappointing behavior of XQuery joins, whose semantics depends on the order in which variables are bound. The following example shows how *TupSort* can be used to preserve the order among XML elements.

Example 3.10 Consider the following query:

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

XQuery semantics [FFM+01] 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.

■

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:

- $Att(e)$ is the set of labels of an *Env* structure e;
- $FV(exp)$ is the set of free variables occurring in an algebraic expression exp;
- $\$x$ denotes a variable

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.

$$\sigma_{P_1 \wedge P_2}(e) \equiv \sigma_{P_1}(\sigma_{P_2}(e))) \tag{4.1}$$

$$\sigma_{P_1}(\sigma_{P_2}(e)) \equiv \sigma_{P_2}(\sigma_{P_1}(e)) \tag{4.2}$$

$$\sigma_{P_1}((e_1) \bowtie_{P_2} (e_2)) \equiv (e_1) \bowtie_{P_1 \wedge P_2} (e_2) \tag{4.3}$$

$$(e_1) \bowtie_{P_1 \wedge P_2} (e_2) \equiv (\sigma_{P_1}(e_1)) \bowtie_{P_2} (e_2) \qquad \text{if } P_1 \text{ applies to } e_1 \text{ only} \tag{4.4}$$

$$(e_1) \bowtie_{Pred} (e_2) \equiv (e_2) \bowtie_{Pred} (e_1) \tag{4.5}$$

4.2 *path* decompositions

path is the most important operator in the algebra; therefore, its evaluation affects the efficiency of the whole query processing. An efficient evaluation of a *path* operation depends on the ability of the query processor to simplify the path expression, and to exploit the presence of access support structures; in particular, the use of appropriate indexes can dramatically speed up the evaluation. Thus, the ability to decompose a complex filter, which may consist of deep paths and require iteration on large collections (for the purpose of binding), into smaller ones, whose paths can be mapped into existing indexes, is crucial.

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 ■

Proposition 4.3 *Horizontal decomposition of conjuncted 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 disjuncted 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 following example shows the use of horizontal decomposition laws.

Example 4.5 ■

4.3 Nested queries equivalences

This Section presents some equivalence rules that can be used to transform dependency joins into $TupJoin$ operations. These rules are not intended to be exhaustive, nor to be the most efficient transformations; they just transform $DJoins$ induced by nested queries into more tractable joins.

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.

Selection 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 *Sigma Dependency*

$$e_1 < (path_f(return_{of}(\sigma_{Pred(\$X,\$Z,\$Y)}(path_{f_1}(db))))) >$$

$$\equiv$$

$$e_1 \bowtie_{Pred_{Glob}(\$X,\$Z)} (path_{f'}(return_{fo'}(\sigma_{Pred_{Loc}(\$Y,\$Z)}(path_{f_1}(db)))))$$
$$where$$

- $fo' = of, z_1[\$z_1], \ldots, z_k[\$z_k]$
- $f' = f, (\_, \_, in)env[(/, \_, in)tuple[(/, \_, in)z_1[(/, z_1, =)\_[\emptyset], \ldots, (/, \_, in)z_k[(/, z_k, =)\_[\emptyset]]]]]$

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

By applying this transformation the dependency is brought out of the inner query.

Example 4.7 ■

return **dependency** In this form of dependency the output filter of contains some external variables $\$X$; these variables can be just projected, or deeply nested into a complex skeleton. In any case, the output filter $of(\$X)$ cannot be decomposed into a local part and a global one; therefore, a different strategy must be adopted. Unlike GOM and OQL, *return* dependencies can still be simplified; the main idea is to *copy* the left part of the $DJoin$ into the nested query, hence transforming it into a constant query, and to combine results by using an equality predicate. This is feasible since the translation scheme of Xtasy algebra is different from that of GOM.

Proposition 4.8 *return dependency*

$$e_1 < (path_f(return_{of(\$X)}(\sigma_{Pred()}(path_{f_1}(db))))) >$$

$$\equiv$$

$$e_1 \bowtie_{\$X=\$X} path_{f,bind}(return_{of'}(\pi(\$X)(e_1) \bowtie_{true} (\sigma_{Pred}(path_{f_1}(db))))$$
$$where$$

- $bind \equiv (\_,\_,in)env[(/,\_,in)tuple[(/,\_,in)x_1[(/,x_1,=)\_[\emptyset], \ldots,(/,\_,in)x_k[(/,x_k,=)\_[\emptyset]]]]]$
- $of' \equiv of(\$X), x_1[\$x_1], \ldots, x_k[\$x_k]$

if $FV(f) \cap Att(e_1) = \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $FV(Pred) \cap Att(e_1) = \emptyset$, $\$X \subseteq Att(e_1)$

Example 4.9 ■

path **dependency** In this form of dependency, the input filter of the inner query is applied to variables coming from the outer query. The transformation is quite similar to that of the *return* dependency.

Proposition 4.10 *path dependency*

$$e_1 < (path_f(return_{of}(\sigma_{Pred}(path_{f_1}(db))))) >$$

$$\equiv$$

$$e_1 \bowtie_{\$X=\$X} (path_{f,bind}(return_{of'}(\sigma_{Pred}(path_{f_1}(e_1)))))$$
$$where$$

- $bind \equiv (\_,\_,in)env[(/,\_,in)tuple[(/,\_,in)x_1[(/,x_1,=)\_[\emptyset], \ldots,(/,\_,in)x_k[(/,x_k,=)\_[\emptyset]]]]]$
- $of' \equiv of(\$X), x_1[\$x_1], \ldots, x_k[\$x_k]$

if $FV(f) \cap Att(e_1) = \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $FV(Pred) \cap Att(e_1) = \emptyset$, $FV(db) = \$X \subseteq Att(e_1)$

Example 4.11 ■

5 Expressive Power

In this section a brief overview of the expressive power of the Xtasy algebra will be given. First, a relational completeness result will be shown; then, we will characterize the class of XQuery queries that can be represented in the algebra.

5.1 Relational Completeness

Current XML query languages can be used to query not only semistructured or irregular data, but also regular data, such as relational databases. This usually happens in integration systems, which provide a uniform XML view of multiple heterogeneous data sources. Therefore, it is important to show that the Xtasy algebra can be used to query relational data sources.

Theorem 5.1 *Completeness for relational algebra with aggregates*

There exists an encoding scheme $\mathcal{M}$ of relational data into XML data, and an encoding scheme $\mathcal{M}'$ of relational algebraic expressions extended with aggregation into Xtasy algebraic expressions, such that for any relational database db and for any correct relational algebraic expression Q on db:

$\mathcal{M}(Q(db)) \equiv \mathcal{M}'(Q)(\mathcal{M}(db))$

5.2 Representation and Translation of XQuery Queries

As in common object-oriented query languages, XQuery queries translation into algebraic expression is performed in two pahses. During the first phase, common syntactical transformations are applied to the query tree; in particular:

1. any binder occurring in the where clause (e.g., `SOME $y IN ...`) is moved to the for clause, and corresponding quantified predicates are introduced in the where clause;
2. any path expression occurring free in the where clause or in the return clause is bound to a variable, and the corresponding binder is introduced in the for - let clauses;
3. nested queries, occurring in any clause, are bound to variables and corresponding binders are introduced in the define clause (similar to the define clause of GOM);
4. finally, common subexpressions are factorized.

The following example shows the transformations applied to a sample query.

Example 5.2 Consider the following query:

```
FOR $b in book,
    $t in $b/title
WHERE EVERY $a in $b/author SATISFIES $a != "Vassilis Christophides"
      AND $b/publisher IN Q
RETURN <entry> $t </entry>
```

This query returns the title of each book not written by Vassilis Christophides, whose publisher is contained into the result of a nested query Q. By applying the above transformations, this query is transformed as follows:

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

```
LET $p = $b/publisher
DEFINE $_var = Q
WHERE EVERY $a SATISFIES $a != "Vassilis Christophides"
      AND $p IN $_var
RETURN <entry> $t </entry>
```

■

After the first phase, for and let clauses are examined in order to build *filter-like* path trees. This is performed by transforming each path expression into a filter-like path, and then by merging together paths referring to the same persistence root.

Finally, the return clause is scanned in order to build corresponding output filters.

Thus, the output of this phase is a query satisfying the following grammar:

```
Q ::= BWSR
B ::= (F|D)+
F ::= for input_filter do
L ::= define var = Q do
W ::= where BoolExp do
S ::= sort_by sort_criteria
R ::= return output_filter
```

Given a query conforming to the previous grammar, its algebraic representation is obtained by applying the following translation scheme.

$[\![BWSR]\!]\rho \equiv [\![R]\!]([\![S]\!]([\![W]\!]([\![B]\!]\rho)))$

$[\![R]\!]\rho \equiv [\![return \;\; output\_filter]\!]\rho \equiv return_{output\_filter}(\rho)$

$[\![S]\!]\rho \equiv [\![sort\_by \;\; sort\_criteria]\!]\rho \equiv Sort_{sort\_criteria}(\rho)$

$[\![W]\!]\rho \equiv [\![where \;\; BoolExp]\!]\rho \equiv \sigma_{BoolExp}(\rho)$

$[\![B]\!] \equiv [\![b_1, \ldots, b_n]\!]\rho \equiv [\![b_1]\!]\rho < [\![b_2]\!]\rho < \ldots < [\![b_n]\!]\rho > \ldots >>$ where:

- $[\![for \;\; input\_filter \;\; do]\!]\rho \equiv path_{input\_filter}(\rho)$ and
- $[\![define \;\; var = Q \;\; do]\!]\rho \equiv path_{(\_,var,=)\_[\emptyset]}([\![Q]\!]\rho)$

Given any query in normal form, it can be translated into an equivalent algebraic expression.

Theorem 5.3 *There exists a query translation scheme $\mathcal{T}$, such that, for each query Q in normal form, $\mathcal{T}(Q)$ is equivalent to Q.*

6 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 The YAT query algebra has been developed in the context of the YAT project [CDSS98]. YAT is an integration system based on a semistructured tree data model, endowed with the ability to specify references as well as recursive structures. Its query algebra, largely based on the object-oriented algebra described in [CM93], manipulates relational-like intermediate structured. 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 it can perform grouping and sorting operations.

Although very powerful, *bind* cannot directly evaluate regular path expression or XPath patterns; indeed, recursive path expressions can be encoded in the YAT algebra only by using recursive algebraic calls.

SAL SAL [BT99] 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 [JLST01] is a query algebra for XML data based on an ordered data model. Unlike YAT and SAL, the TAX algebra directly manipulates tree data 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 very promising, the optimization properties of TAX are not clear.

Other algebras In [BMR99] 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., $A \bowtie B \neq B \bowtie A$. The bad consequences of this design choice are clear.

7 Conclusions and Future Work

This paper described 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 have to explore 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, in the light of the recent proposals of global ordering emerged in the W3C XML Query Working Group discussions.

8 Acknowledgements

Authors thank Dario Colazzo for his precious suggestions.

References

[ASL89] A. M. Alashqur, Stanley Y. W. Su, and Herman Lam. Oql: A query language for manipulating object-oriented databases. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 433–442. Morgan Kaufmann, 1989.

[BMR99] David Beech, Ashok Malhotra, and Michael Rys. A formal data model and algebra for xml. Note to the W3C XML Query Working Group, 1999.

[BOS94] Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors. *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing. Springer, 1994.

[BT99] Catriel Beeri and Yariv 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.

[CCF+01] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, June 2001. W3C Working Draft.

[CCMS98] Vassilis Christophides, Sophie Cluet, Guido Moerkotte, and Jérôme Siméon. Optimizing generalized path expressions using full text indexes. *Networking and Information Systems Journal*, 1(2):177–194, 1998.

[CCS98] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. Semistructured and Structured Integration Reconciled: YAT += Efficient Query Processing. Technical report, INRIA, Verso database group, November 1998.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Technical report, World Wide Web Consortium, November 1999. W3C Recommendation.

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1998. ACM Press.

[CG01] Luca Cardelli and Giorgio Ghelli. A query language for semistructured data based on the ambient logic. In *Proceedings of the 10th European Symposium on Programming, ESOP 2001, Held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001*, pages 1–22, 2001.

[CGLV01] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-Based Query Answering and Query Containment over Semistrctured Data. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01), Frascati, Rome, September 8-10, 2001*, 2001.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Beeri et al. [BOS94], pages 226–242.

[CMS01] Dario Colazzo, Paolo Manghi, and Carlo Sartiani. Xtasy: A typed xml database management system. Available at `http://www.di.unipi.it/ sartiani/papers/mementomori.pdf`, 2001.

[CRF00] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 2000.

[DT01a] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001), Rome, Italy, September 15, 2001*, 2001.

[DT01b] Alin Deutsch and Val 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.

[FFM+01] Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Michael Rys, Jerome Simeon, and Philip Wadler. XQuery 1.0 Formal Semantics. Technical report, World Wide Web Consortium, June 2001. W3C Working Draft.

[FM00] Thorsten Fiebig and Guido Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proceedings of the third International Workshop WebDB 2000*, pages 125–136, 2000.

[FR00] Mary Fernandez and Jonathan Robie. XML Query Data Model. Technical report, World Wide Web Consortium, May 2000. W3C Working Draft.

[GW97] Roy Goldman and Jennifer 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.

[HP00] Haruo Hosoya and Benjamin C. Pierce. Xduce: A typed xml processing language (preliminary report). In *Proceedings of the Third International Workshop on the Web and Databases, WebDB 2000, Adam's Mark Hotel, Dallas, Texas, USA, May 18-19, 2000, in conjunction with ACM PODS/SIGMOD 2000*, pages 111–116, 2000.

[JLST01] H.V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith 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.

[KM00] Carl-Christian Kanne and Guido 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.

[LMS+93] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The aqua data model and algebra. In Beeri et al. [BOS94], pages 157–175.

[Mar01] Jose M. Martinez. Overview of the mpeg-7 standard (version 5.0). Technical report, International Organisation for Standardisation, 2001.

[med] `http://xmlmarc.stanford.edu/MLA2001/medlane.html`.

[MFK+00] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Don Olteanu. Agora: Living with xml and relational. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young 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.

[MSF01] Steve Muench, Mark Scardina, and Mary Fernandez. XPath Requirements Version 2.0. Technical report, World Wide Web Consortium, February 2001. W3C Working Draft.

[MW98] Jason McHugh and Jennifer Widom. Compile-time path expansion in Lore. Technical report, Stanford University Database Group, November 1998.

[QRSU95] 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.

[Sim99] Jérôme Siméon. *Intégration de sources de données hétérogènes*. PhD thesis, Université Paris XI, 1999.

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

A Formalization

A.1 *Env* and tuples operations

Four basic operations are defined on *Env* structures and tuples.

1. $t.A = t_j$ where $A = label_j$ (where t is a tuple) (field extraction)
2. $t.\vec{A} = \{t_{i1}, \ldots, t_{ip}\}$ where $\vec{A} = (label_{i_1}, \ldots, label_{i_p})$ (repeated field extraction)
3. $t \downarrow \vec{A} = [label_{i1} : t_{i1}, \ldots, label_{ip} : t_{ip}]$ where $\vec{A} = (label_{i_1}, \ldots, label_{i_p})$
4. $\bullet$, a concatenation operator between tuples (known as *tup_concat* in other algebras).

A.2 Support operators

1. $e[x] = \{[x : t] \quad | \quad t \in e\}$
2. $child(t) =$
   (a) if $t = v_B$, then $child(t) = \{\}$
   (b) if $t = \_[t_1, \ldots, t_n]$, then $child(t) = \{t_i \quad | \quad i \in 1, \ldots, n\}$
3. $descendant(t) = child(t) \cup \bigcup_{t_i \in child(t)} descendant(t_i)$
4. $self(t) = \{t_1, \ldots, t_n \quad | \quad t = t_1, \ldots, t_n\}$

5. $self - descendant(t) = self(t) \cup descendant(t)$

6. $nav(op)(label)(t) =$

   (a) if $op = (\_)$, then $nav(op)(label)(t) = \{t_i \quad | \quad t = t_1, \ldots, t_n \wedge label(t_i) = label\}$

   (b) if $op = (/)$, then $nav(op)(label)(t) = \{t_j^{'} \quad | \quad t = t_1, \ldots, t_n \wedge \exists i \in 1, \ldots, n : t_j^{'} \in child(t_i) \wedge label(t_j^{'}) = label\}$

   (c) if $op = (\forall/)$, then

$$nav(op)(label)(t) = \begin{cases} nav(/)(\_)(t) & \text{if } \forall t^{'} \in child(t) : label(t^{'}) = label \\ \{\} & \text{otherwise} \end{cases}$$

   (d) if $op = (//)$, then $nav(op)(label)(t) = \{t_j^{'} \quad | \quad t = t_1, \ldots, t_n, \exists i \in 1, \ldots, n : t_j^{'} \in self - descendant(t_i) \wedge label(t_j^{'}) = label\}$

   (e) if $op = (\forall//)$, then

$$nav(op)(label)(t) = \begin{cases} nav(//)(\_)(t) & \text{if } \forall t^{'} \in self - descendant(t) : label(t^{'}) = label \\ \{\} & \text{otherwise} \end{cases}$$

A.3 Basic Operators

Map $\chi_f(e) = \{f(t) \quad | \quad t \in e\}$

Join $e_1 \bowtie^f_{Pred} e_2 = \{f(t_1, t_2) \quad | \quad t_1 \in e_1 \wedge t_2 \in e_2 \wedge Pred(t_1, t_2)\}$

TupJoin $e_1 \bowtie_{Pred} e_2 = \{t_1 \bullet t_2 \quad | \quad t_1 \in e_1 \wedge t_2 \in e_2 \wedge Pred(t_1, t_2)\}$

DJoin $e_1 < e_2 >= \{y \bullet x \quad | \quad y \in e_1, x \in e_2(y)\}$

Selection $\sigma_{Pred}(e) = \{t \quad | \quad t \in e, Pred(t)\}$

Projection $\pi_{\underset{A}{\rightarrow}}(e) = \{t \downarrow \vec{A} \quad | \quad t \in e\}$

GroupBy $GroupBy_{g;A;f_1;f;\theta}(e) = \{y.A \bullet [g : G] \quad | \quad y \in e, G = f(\{x \quad | \quad x \in e, f_1(x)\theta f_1(y)\})\}$ where $A \subseteq Att(e)$ and $y \notin Att(e)$

Sort $Sort_{Pred}(e) = \langle t_1, \ldots, t_n \rangle$ such that $t_i \in e \ \ (i = 1, \ldots, n)(\forall t \in e \exists i \in [1, n] : t = t_i)(\forall i \in [1, n-1] : Pred(t_i, t_{i+1})$ where $Pred$ is an ordering predicate.

TupSort $TupSort_{(x_1, \ldots, \$x_n)}(e) = Sort_{<_{Tup}(\$x_1, \ldots, \$x_n)}(e)$ where:

$$\begin{aligned} <_{Tup} (\$x_1, \ldots, \$x_n)(u, v) = \quad & (pos(u.\$x_1) < pos(v.\$x_1)) \vee \\ & (pos(u.\$x_1) = pos(v.\$x_1) \wedge pos(u.\$x_2) < pos(v.\$x_2)) \vee \ldots \vee \\ & (pos(u.\$x_1) = pos(v.\$x_1) \wedge \ldots \wedge pos(u.\$x_{n-1}) = pos(v.\$x_{n-1}) \wedge \\ & pos(u.\$x_n) < pos(v.\$x_n)) \end{aligned}$$

A.4 Path

Input filters grammar

(1) $F ::= F_1, \ldots, F_n \quad | \quad F_1 \vee \ldots \vee F_n \quad | \quad (op, var, binder)label[F] \quad | \quad \emptyset$
(2) $op \in \{/, //, \_, \forall/, \forall//\}$
(4) $var \in label \cup \{\_\}$
(5) $binder \in \{\_, in, =\}$

$path_f(t) =$

1. if $f = f_1, \ldots, f_m$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_{f_1}(t)\ TupJoin(true) \ldots TupJoin(true)\ path_{f_m}(t)$
2. if $f = f_1 \vee \ldots \vee f_m$, then $path_f(t) = path_{f_1}(t)\ OuterUnion \ldots OuterUnion\ path_{f_m}(t)$
3. if $f = (\_, \_, binder)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{\}$;
4. if $f = (\_, \_, binder)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_F(nav(\_)(label)(t))$;
5. if $f = (op, l, in)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = nav(op)(label)(t)[l]$;
6. if $f = (op, l, =)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{[l : nav(op)(label)(t)]\}$;
7. if $f = (op, l, in)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \bigcup_{t_i \in nav(op)(label)(t)} \{[l : t_i]\} TupJoin(true) path_F(t_i)$
8. if $f = (op, l, =)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{[l : nav(op)(label)(t)]\}$ $TupJoin(true)\ path_F(nav(op)(label)(t))\}$;
9. if $f = (op, \_, \_)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{\}$;
10. if $f = (op, \_, \_)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_F(nav(op)(label)(t))$;
11. $path_\emptyset(t) = \{\}$;

A.5 Return

Output filters grammar

(1)$OF ::= OF_1, \ldots, OF_n \quad | \quad label[OF] \quad | \quad @label[val] \quad | \quad val$
(2)$val ::= v_B \quad | \quad var$

$return_{fo}(e) =$

1. if $fo = v_B$, then $return_{fo}(e) = \bigcup_{i=1}^{n} v_B$ where $e = \{t_1, \ldots, t_n\}$;
2. if $fo = var$, then $return_{fo}(e) = \{t.var \quad | \quad t \in e\}$;
3. if $fo = @label[val]$, then $return_{fo}(e) = \bigcup_{i=1}^{n} @label[return_{val}(\{t_i\})$ where $e = \{t_1, \ldots, t_n\}$;
4. if $fo = label[fo']$, then $return_{fo}(e) = \bigcup_{i=1}^{n} label[return_{fo'}(\{t_i\})$ where $e = \{t_1, \ldots, t_n\}$;
5. if $fo = fo_1, \ldots, fo_n$, then $return_{fo}(e) = \bigcup_{i=1}^{n} return_{fo_1}(\{t_i\}), \ldots, return_{fo_n}(\{t_i\})$ where $e = \{t_1, \ldots, t_n\}$.