On Wrapping Query Languages and Efficient XML Integration*

Vassilis Christophides
Institute of Computer Science
FORTH, P.O. Box 1385
Heraklion, Greece
christop@csi.forth.gr

Sophie Cluet
INRIA Rocquencourt
BP 105, 78153
Le Chesnay Cedex, France
Sophie.Cluet@inria.fr

Jérôme Siméon
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ, USA
simeon@research.bell-labs.com

Abstract

Modern applications (Web portals, digital libraries, etc.) require integrated access to various information sources (from traditional DBMS to semistructured Web repositories), fast deployment and low maintenance cost in a rapidly evolving environment. Because of its flexibility, there is an increasing interest in using XML as a middleware model for such applications. XML enables fast wrapping and declarative integration. However, query processing in XML-based integration systems is still penalized by the lack of an algebra with adequate optimization properties and the difficulty to understand source query capabilities. In this paper, we propose an algebraic approach to support efficient XML query evaluation. We define a general purpose algebra suitable for semistructured or XML query languages. We show how this algebra can be used, with appropriate type information, to also wrap more structured query languages such as OQL or SQL. Finally, we develop new optimization techniques for XML-based integration systems.

1 Introduction

XML [6] is becoming widely used for the development of Web applications that require data integration (Web portals, e-commerce, etc). Although fashion surely accounts for some of XML's popularity, it is also justified on technical grounds. XML enables easy wrapping of external sources and declarative integration, thus allowing fast deployment and cheap maintenance of applications. Still, XML-based systems are not yet as efficient as traditional integration software [39, 8, 40, 26, 22, 7]. In this paper, we address this issue.

Let us consider an example to motivate the use of XML technology and the improvements we propose. In this example, we plan to build a Web site providing access to commercial information about cultural goods (e.g., www.christies.com). For this application, we need to integrate two sources: one, highly structured, is an object database containing trading information; the other is a partially structured document repository supporting full-text queries, that contains descriptive information about artistic work. Figure 1 shows some sample XML data exported from our sources.

```
<object id="a1" class="artifact">
  <tuple>
    <title> Nympheas </title>
    <year> 1897 </year>
    <creator> Claude Monet
    </creator>
    <price> 10.000.000 </price>
    <owners refs = "p1 p2 p3"/>
  </tuple>
</object>
.....
<object id="p3" class="person">
  <tuple>
    <name> Doctor X </name>
    <auction> 10.1500.000
    </auction></tuple>
</object>
```

```
<work>
    <artist> Claude Monet </artist>
    <title> Nympheas </title>
    <style> Impressionist </style>
    <size> 21 x 61 </size>
    <cplace>Giverny</cplace>
</work>
....
<work>
    <artist> Claude Monet </artist>
    <title> Waterloo Bridge </title>
    <style> Impressionist </style>
    <size> 29.2 x 46.4 </size>
    <history>Painted with
      <technique> Oil on canvas
      </technique> in ...
</work>
```

Figure 1: Sample XML Data for Cultural Goods

There are several advantages in building this application with XML. First, due to its flexible data model, XML can represent both structured and semistructured information (see Figure 1). Second, it is easy to convert any data into XML, and to do so in a generic fashion (i.e., independently of the source schema). Third, several languages support declarative integration of XML data (e.g., MSL [31], StruQL [17] or YAT$_L$ [13]). Finally, being a standard, XML facilitates interoperability. Yet, query processing in XML-based integration systems raises some hard issues.

- *Wrapping type information.* There are certainly many reasons why preserving type information is useful, but it is particularly important for query optimization [20]. Although most data management systems can now export data in XML, they usually don't provide the corresponding type information. This is mostly because XML's current form of typing (i.e., DTDs [6]) is not sufficient to capture rich type systems (e.g., an object database schema)

*Project supported by OPAL (Esprit IV project 20377) and AQUARELLE (Telematics Application Program IE-2005).

MOD 2000, Dallas, TX USA

or, conversely, partially structured documents (e.g., in Figure 1, `works` might come with mandatory elements as well as elements not known in advance, like `history` or `cplace`). Several recent proposals (notably XML Schema [38]) are studying this issue, but no definitive standard is available yet. In [13], we introduced a type system, suitable to represent any mix of well-formed and valid XML data, that we will use in the rest of paper.

- *Wrapping source query capabilities.* Internet sources usually do not export data but, instead, provide query facilities. Thus, in order to integrate them, one needs to understand their "query language". This is also important for performance reasons: by pushing the processing to the sources as much as possible, the application avoids massive data transfers and reduces XML conversion overhead. The only technique proposed so far and that would be appropriate for XML, comes from the TSIMMIS system: query templates [33] are used to describe source capabilities. However, an exhaustive description of sources capabilities (i.e., find all possible queries given a schema) is not feasible with such templates. Moreover these imply a costly *ad hoc* development, in order to wrap an appropriate set of queries for each application.

- *Processing XML queries efficiently* in an integration context remains an open problem. A well-understood algebra that supports the peculiarities of XML languages is missing. Moreover, we need the ability to exploit partial type information and heterogeneous source capabilities.

In this paper, we propose an algebraic framework and optimization techniques to address the last two issues. More precisely, we make the following contributions:

An algebra for XML. We introduce an operational model based on a general-purpose algebra for XML. This algebra is expressive enough to capture most of the semantics of existing semistructured/XML or structured query languages.

A source description language. We show how this algebra can be used to wrap full text queries but also structured query languages such as OQL or SQL in a complete (i.e., as a query language and not as a set of queries) and generic (i.e., with no effort required from the application developer) way.

Query processing techniques. We show that our algebra is appropriate to optimize integration applications. Notably, we introduce new rewriting techniques for query composition, investigate the impact of type information during query processing and illustrate how query evaluation can take advantage of source query capabilities.

```
-------------------------------------------------------
logos{simeon}: o2-wrapper -server gringos.inria.fr \
                          -system cultural         \
                          -base art                \
                          -port 6066
 o2-wrapper is running at logos.inria.fr:6066
logos{simeon}:
-------------------------------------------------------
sappho{christop}: xmlwais-wrapper                \
   -directory ~christop/wais-sources/museum.src \
   -port 6060
 xmlwais-wrapper is running at sappho.ics.forth.gr:6060
sappho{christop}:
-------------------------------------------------------
cosmos{cluet}: yat-mediator -port 6666
  yat-mediator is running at cosmos.inria.fr:6666
yat> connect o2artifact logos.inria.fr:6066;
yat> connect xmlartwork sappho.ics.forth.gr:6060;
yat> import o2artifact;
yat> import xmlartwork;
yat> load "/u/cluet/YAT/view1.yat";
```

Figure 2: Installing Wrappers and Mediators

The paper is organized as follows. Section 2 illustrates the advantages of XML integration by explaining the different steps required to build our example application with YAT, our home-brewed integration system. This section also recalls the specifics of the type system we are using. Section 3 introduces our algebra. The description language to wrap source query languages is presented in Section 4. We present the optimization techniques in Section 5 and conclude in Section 6.

2 XML integration with YAT

The YAT System is a semistructured data conversion system [13, 36] that we are currently turning in to a full-fledged XML integration system. It relies on a library of generic wrappers and a declarative integration language called YAT_L. Figure 2 illustrates the three steps required to setup our application example with YAT:

- *simeon* wraps the O_2 object database. For this, he simply needs to run the `o2-wrapper` program that can export structural information from any O_2 database (e.g., the `art` database) as well as the system query capabilities (i.e., it wraps OQL, as we will see in Section 4).

- *christop* wraps the cultural source with another generic wrapper. The `xmlwais` wrapper understands XML data, typed with our type system and full-text indexed by Wais [34]. It expects as parameter a standard Wais source configuration file (e.g., `museum.src`).

- *cluet* runs a `yat` mediator, connects both wrappers using the port numbers given by her fellow developers, imports the structural and query capabilities of the two connected system and loads her favorite integration program (e.g., `view1.yat`).

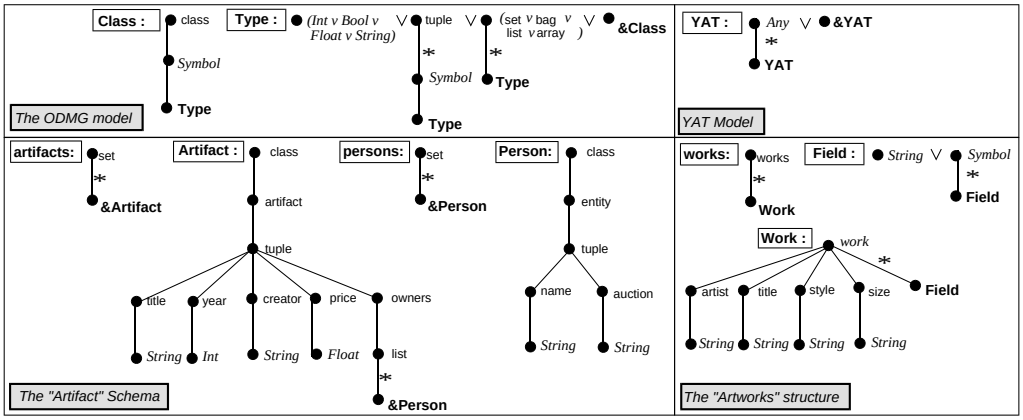

Figure 3: O_2, XML-Wais and YAT mediator structural metadata

Before taking a closer look at the integration program, we first give the structural information exported by each wrapper. Note that for interoperability reasons, wrappers and mediators communicate data, structures and operations in XML.

The YAT type system can represent structural information at various levels of genericity (model, schema, data). The relationship between these levels is captured through an *instantiation* mechanism that we recall here briefly (see [13] for more details). Figure 3 shows a graphical representation of the YAT model along with the type information imported by our wrappers.

The left hand-side of the figure represent the O_2 data model, that conforms to the ODMG standard [9], and the schema of our `art` database example. Note that (i) bold fonts denote pattern (i.e., tree) identifiers, (ii) the `&` symbol denotes references, (iii) the $\star$ and $\vee$ symbols denote respectively multiple occurrences and alternatives. Thus, an O_2 type is either an atomic type, a tuple, a collection or a reference to a class pattern. A tuple pattern is in turn a collection of subpatterns, each associating an attribute name (*Symbol*) to its value. Below, the class `Artifact` is a concrete instantiation of a `Class`, whose value is a tuple with attributes `title`, `year`, etc. The lower right part of the figure represents the information exported by the `xmlwais` wrapper. Each document contains mandatory information (`artist`, etc), possibly followed by any additional `Fields`. This illustrates the ability of YAT type system to capture partially structured information.

Last, the top right part of Figure 3 shows a representation of YAT (meta)model, that captures all patterns. One important property is that the O_2 model, `Artifacts` schema, and `Artworks` structure are recognized as instances of this almighty model (in fact, we have `Artifact <: ODMG <: YAT`). We will see in Section 4 that query languages wrapping will also take advantage of this mechanism.

Integration programs in declarative languages are usually composed of a sequence of rules or queries [31, 17, 13], whose partial results are connected together through Skolem functions. We give bellow an example of a YAT_L query [19, 37], from our integration program `view1.yat`. This query construct a collection of documents (`artworks`), one per known artwork, each combining the information available in our two sources.

```
artworks() :=
    MAKE doc * &artwork($t,$c) :=
        work [ title: $t, artist: $a,
               year: $y, price: $p,
               style: $s, size: $si,
               owners *$o, more: $fields ]
    MATCH artifacts WITH
        set *class: artifact:
                tuple [ title: $t, year: $y,
                        creator: $c, price: $p,
                        owners: list *class: person:
                            tuple [ name: $o,
                                    auction: $au] ],
                works WITH
        works *work [ artist: $a,
                      title: $t', style: $s,
                      size: $si, *($fields) ]
    WHERE $y > 1800 AND $c = $a AND $t = $t'
```

This query consists of three clauses. The **MATCH** clause performs pattern-matching: filters are used to navigate through the structure of data and to bind variables to information of interest (e.g., the artifact's title to variable `$t`, the list of optional XML elements to `$fields`). YAT_L's filtering mechanism relies on instantiation: if a tree is instance of a filter, then one can deduce a mapping between node values and variables. Otherwise, a type error occurs. Note that for unambiguous filters (i.e., involving unambiguous regular expressions), this can be done in polynomial time [4]. The **WHERE** clause fulfills the usual function. The **MAKE** clause constructs the result by creating a new pattern with the values returned by

the previous clauses. In the example, we build a new `artwork` tree for each distinct artifact and group these subtrees under the `doc` node. Here, `artwork($t,$c)` is a Skolem function, creating a new tree identifiers for each distinct values of title and creator. Using Skolem functions allow us to identify (sub)trees and, thus, to create references. Note that the type information provided by the wrappers and by the YAT$_L$ program can be used to guide the integration specification, check application consistency or notify the integration administrator about source modifications.

Technical challenges in query processing. This illustrates the simplicity of XML-based integration. Apart from the quality of structural descriptions provided by YAT, other semistructured/XML systems (like TSIMMIS [32] or MIX [3]) would offer similar functionalities. Still, we have to evaluate user queries in an efficient way. As an invitation to proceed further, assume a user, after noticing some artworks with a creation place (`cplace`), issues the following query:

Q1: *What are the artifacts created at "Giverny"?*

MAKE $t
MATCH artworks **WITH** doc.work.[title.$t,
more.cplace.$cl]
WHERE $cl = "Giverny"

In order to process **Q1**, we need to address several problems: how to compose it with the view definition (note that **Q1** accesses the semistructured fields of artwork documents), how to understand that only the XML-Wais source is needed to answer the query and how to exploit the Wais textual queries to avoid downloading all the documents.

3 YAT operational model and algebra

The choice of the operational model is essential: in the remainder of the paper, it will be used for the description of source capabilities as well as for query optimization. Moreover, it must support the following requirements:

Expressive power. It must capture the evaluation of existing languages, along with their XML-specific features. Notably complex pattern matching primitives with ordered navigation (like in XQL [35] or YAT$_L$), recursion and object creation.

Support for flexible typing. XML favors flexibility and most XML query languages are not typed. Yet, we also need to wrap structured languages. Thus, the operational model must support both flexible type filtering (for Lorel[1] or XML-QL[16]) and more strict forms of typing (for OQL [9]).

Support for optimization. Of course, we also need an algebra equipped with a number of equivalences offering interesting optimization opportunities.

We propose a operational model based on a functional approach, and a fixed set of predefined functions – the so-called YAT XML algebra. The model allows composition, function calls, and recursion. Note that except for Skolem functions, all other functions are without side-effects. The algebra itself is inspired from the object algebra of [14]. In this section, we present the newly introduced operators, required to deal with tree structures, and only briefly recall the others. We show how queries are translated in this operational model. Finally, we give an overview of alternative algebras.

3.1 YAT XML algebra

One of the main characteristics of XML data is that, like objects, it can be arbitrarily nested. Thus, we adopt a technique similar to that used for object-oriented algebras. Starting from an arbitrary XML structure, we apply an operator, called *Bind*, whose purpose is to extract the relevant information and produce a structure, called *Tab*, comparable to a ¬1NF relation. On these *Tab* structures, we can then apply the classical operators, such as *Join*, *Select*, *Project*, etc. Finally, an inverse operation to *Bind*, called *Tree*, can be used to generate a new nested XML structure.

The Bind operator extracts data from an input tree according to a given filter (i.e., a tree with distinct variables). It produces a table that contains the variable bindings resulting from the pattern-matching. On Figure 4, the *Bind* operation is applied on the tree representing the XML collection of works, with a filter that binds for each work its title (`$t`), artist (`$a`), style (`$s`), size (`$si`) and optional elements (note that, being on the edge, variable `$fields` will contain the *collection* of such elements). Note the similarity between the *Tab* structure and a ¬1NF relation. The *Bind* operator supports type filtering, vertical and horizontal navigation (through regular expressions – see `$fields` in our example). It can be expensive to evaluate, but we will see in Section 5.1 how to rewrite a *Bind* into more simple operations.

The Tree operator is applied on *Tab* structures and returns a collection of trees conforming to some input pattern. On Figure 4, the *Tree* operation is applied on the result of the previous *Bind* (where `F[$t,$a,$s,$si,$fields]` denotes the corresponding filter). The works are grouped according to the artists' names (with the grouping primitive `*($a)`), with each subtree containing the titles of their works.

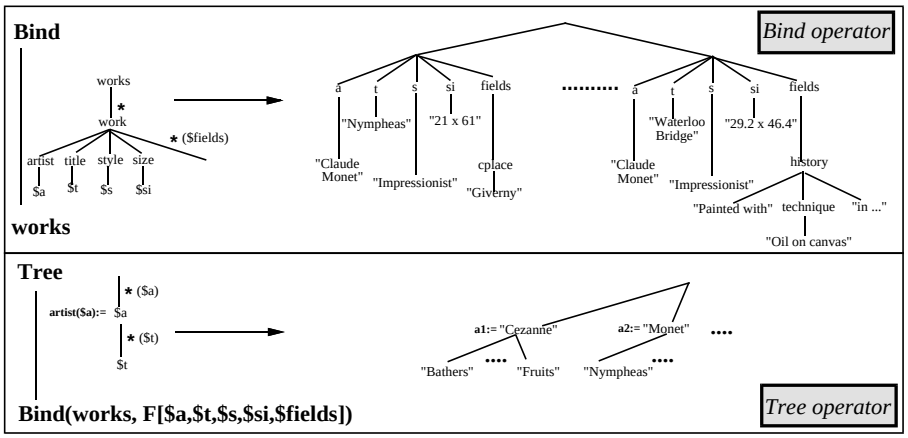

Figure 4: Bind and Tree operators

Skolem functions are used to create new identifiers and perform value assignment. In the previous example, `artist($a):=` creates an identifier for each artist name. Skolem functions do not create values but have side effects on the integrated view (as in [2]) and are somehow orthogonal to the rest of the algebra.

The other operators are those of the object algebra of [14]. *Select*, *Project*, *Join*, *Union*, *Intersection* come from relational. Classical object operations are: *Group*, *Sort*, *Map* and *D-Join* (for dependency join) which is used to navigate within nested collections. Their definition on *Tab* structures rather than collections of tuples is straightforward. We do not recall their definition here, but will explain their use whenever necessary. Except for the *Map*, these operators are always applied on the top level of a *Tab* structure (in a manner similar to the relational algebra). If one needs to go deeper, an extra *Bind* has to be applied.

As most of the algebra is composed of standard operators, we can take advantage of their well-known optimization properties and reuse rewriting techniques proposed in the object context (including relational ones or those for nested queries [14]). We can remark that *Bind* and *Tree* are two frontier operations that isolate XML-specific processing from more standard one. Last, by allowing recursive calls in the algebra (which was not the case of [14]), we can capture generalized path expressions (GPE) [11, 1]. The optimization of GPE is not addressed here (see [12, 20]).

An important aspect is that the YAT algebra is independent of any underlying physical access structure and can be used to reason about the evaluation of XML queries, whether the corresponding XML data are locally stored (e.g., in a document management system or an XML repository) or virtually accessed (e.g., through wrappers as in our context). In Section 5 we will present useful rewritings for both cases.

3.2 YAT$_L$ algebraic translation

Figure 5 shows the algebraic translation of the YAT$_L$ view definition presented in Section 2 and of query **Q1** (translation of other XML query languages would be performed in a similar manner[1]). It has been obtained using the following translation steps:

1. Named documents (e.g., `artifacts`) are the input operations of the algebraic expression.
2. Each **MATCH** statement translates into a *Bind* operation that captures its filtering/binding semantics, and creates a *Tab* structure for further processing.
3. Predicates involving various inputs translate into *Join* operations.
4. Other predicates in the **WHERE** clause translate into *Select* operations.
5. The **MAKE** clause translates into a *Tree* operation.

3.3 Related work

The Lore algebra [27] is a physical algebra, aimed at the optimization involving indexes. SAL [5] is a logical XML algebra, but it does not provide the appropriate expressive power either. The algebra of [18] is both logical and is sufficiently expressive. Yet, the relationship with our algebra is still unclear. For instance, they provide a simpler version of the Bind operator in terms of regular expression matching, while we will see that our more complex Bind can serve in exploiting source capabilities. Compared to object algebras, the *Bind* resembles the *Scan* operator of [15] (minus the condition, plus potentially complex patterns). An object algebra with side-effects operations similar to Skolem functions is presented in [2].

[1] Note that translating some particular features, like recursive structure preservation in XQL, would be more involved.

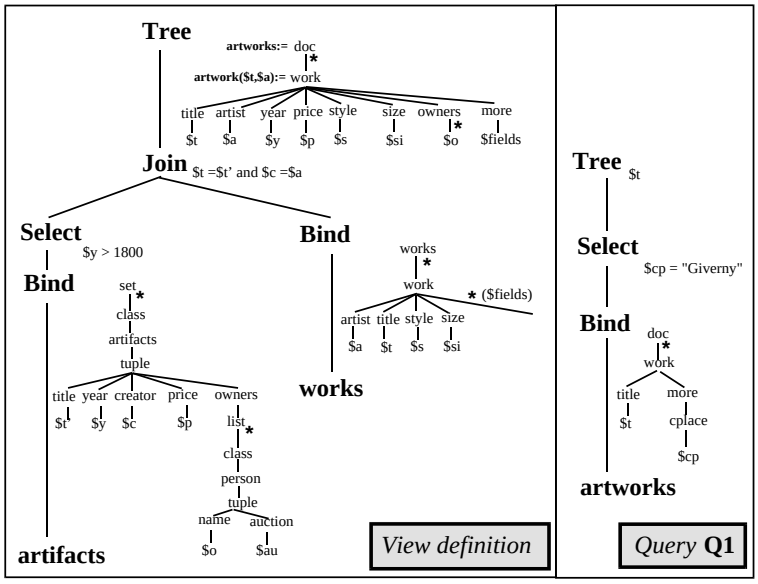

Figure 5: Algebraization of YAT$_L$ queries

4 Wrapping query capabilities

As we explained in Section 2, each wrapper exports its source capabilities. In this section, we explain how this information is communicated to the mediator. Moreover, we show how the combination of our operational model and type system allows to do it at the appropriate level of genericity: from full query languages (e.g., OQL on the ODMG model) to sets of queries (e.g., methods of an O_2 schema, textual predicates on XML elements).

Wrapping source operations in YAT is performed in two steps that concern (i) their signature and (ii) their semantics. The first step is necessary to be able to access the operation. For instance, let us assume that our O_2 schema features a specific method: `current_price` on class Artifact. It can be imported by the O_2 wrapper using the following XML syntax:

```
1   <operation kind="external" name="current_price">
2    <input><value model="Artifact_Schema"
3                  pattern="Artifact"/></input>
4    <output><leaf label=Float /></output>
5   </operation>
```

The `input` and `output` elements contain the signature: `current_price` takes an `Artifact` and returns a `Float`. This declaration is performed automatically by the O_2 wrapper with the help of the O_2 schema manager. Once the method is wrapped, it can be made available at the mediator.

The second step is only required for optimization purposes. In most cases, the wrapper performs both steps automatically. However, for the sources featuring operations not captured by the core operational model, the second step must be done manually. This issue is discussed in Section 5. Now, let us explain more precisely how to use all this to capture OQL and Wais.

4.1 Describing OQL capabilities

We consider here the description of OQL [9]. Obviously, SQL [28] can be described in a similar manner (even-though the wrapper's implementation is more complex due to the non-functional nature of SQL).

Capturing binding capabilities. YAT operational model borrows a large part of OQL algebra [14]. But if YAT captures OQL, the opposite is not true mostly for one reason: OQL binding capabilities are more restricted (e.g., it cannot query schema information). In order to take this restriction into account, we need to distinguish between *Bind* operations that can be actually evaluated by OQL and those that cannot, i.e., we need to understand which are the acceptable filters for OQL. Figure 6 (lines 2 to 33) shows such a specification of valid filters (that we call a *Fmodel*). The O_2 *Fpatterns* are nothing but an XML serialization of the type patterns of Figure 3, possibly annotated with flags (attributes `bind` and `inst`). When present, flags correspond to filter restrictions. A `bind` flag can be used to indicate that the corresponding node cannot contain a variable, or only a tree or label variable. A `inst` flag can be used to indicate that the corresponding label or edge must be completely instantiated (`ground` value) or left unchanged (`none` value). For instance, the filter for O_2 classes (`Fclass`, line 3) imposes that (i) only subtrees corresponding to actual O_2 objects or values can be bound (`bind="tree"`, line 4) (ii) extraction of class schema information is prevented (`bind="none"`,

```
1  <interface name="o2artifact">
2   <fmodel name="o2fmodel">
3   <fpattern name="Fclass">
4    <node label="class" bind="tree">
5    <node label="Symbol" bind="none" inst="ground">
6    <value pattern="Ftype"/></node></node>
7   </fpattern>
8   <fpattern name="Ftype">
9    <union>
10    <leaf label="Int"/>
11    <leaf label="Bool"/>
12    <leaf label="Float"/>
13    <leaf label="String"/>
14    <node label="tuple" col="set" bind="tree">
15     <star inst="ground">
16      <node label="Symbol" bind="none">
17      <value label="Ftype"/></node></star></node>
18    <node label="set" col="set" bind="tree">
19     <star inst="none"><value label="Ftype"/>
20     </star></node>
21    <node label="bag" col="bag" bind="tree">
22     <star inst="none"><value label="Ftype"/>
23     </star></node>
24    <node label="list" bind="tree">
25     <star inst="none"><value label="Ftype"/>
26     </star></node>
27    <node label="array" bind="tree">
28     <star inst="none"><value label="Ftype"/>
29     </star></node>
30    <ref pattern="Fclass"/>
31   </union>
32  </fpattern>
33  </fmodel>
34
35  <operation name="bind" kind="algebra">
36   <input>
37    <value model="o2model" pattern="Type"/>
38    <filter model="o2fmodel" pattern="Ftype"/></input>
39   <output><value model="yat" pattern="Tab"/></output>
40  </operation>
41  <operation name="select" kind="algebra"></operation>
42  <operation name="map" kind="algebra"></operation>
43  <operation name="eq" kind="boolean"></operation>
44 </interface>
```

Figure 6: O_2 Filter patterns and operational interface

line 5) and (iii) the name of the class in a schema specific filter has to be instantiated (`inst="ground"`, line 5).

OQL operations. Figure 6 also shows a large part of the operational interface exported by the O_2 wrapper (lines 35 to 43). Each operation has a `name` (`bind`, `eq`, etc) and a `kind` (`algebra`, `boolean`, `external`, etc).

The first declared algebraic operation is the *Bind* (line 35). Its signature has been *specialized* using the already exported *Fpattern* `Ftype` (line 8). The other algebraic operators that O_2 can evaluate follow (`select`, `map`, etc). We do not need to specialize their signatures as these operations are always applied on a *Tab* structure resulting from a *Bind*, i.e., on collection of tuples containing valid ODMG data. Note that an operation can be pushed only on some data imported by the source or on the result of a previously pushed operation. Furthermore, all the arguments of the operation must be pushable. For instance, a selection can be pushed only with the predicates (e.g., =, <=, etc.)) or functions (e.g., the method `current_price`) that are understood by O_2. In the case of our integration example, the *Bind* and *Select* operations on the left-hand side of Figure 5 can be pushed to O_2 and translated by the wrapper into the following equivalent OQL query:

select t: A.title, y: A.year, c: A.creator, p: A.price,
n: O.name, au: O.auction,
from A in artifacts, O in A.owners
where A.year > 1800

4.2 Describing Wais capabilities

For most sources, one of the basic operation is to ask for an entry point (a relation, a named object, a document, etc). However, even this seemingly simple operation is not always supported. For instance, many Web sites (e.g., search engines) are only accessible through form-based query interfaces and do not export their full content. For these sources, it is capital to understand the operations they supported even if these are not captured by the original YAT algebra.

Another apparently straightforward assumption is that you can retrieve what you query. Again, this is not always true. The Z39.50 [41] protocol (underlying the Wais retrieval engine and which is widely used for digital libraries) is based on attribute/value textual queries. This protocol establishes a clear separation between what you may retrieve and what you may query. For instance, one could specify that only the `artist` and `style` elements can be exported from our XML documents while allowing queries only on the optional fields [29]. This can be captured, thanks to the extensibility of our operational model, by declaring a predicate for each queried field and exporting them to the mediator.

Importing the query capabilities of an XML-Wais source. We now show how to wrap the full-text capabilities of our XML-Wais source ("signature" step), and how to declare a source-specific equivalence ("semantic" step). For the first step, we need to: (i) specify the source *Fpatterns*, (ii) declare support for *Bind* and *Select* operations, and (iii) declare the full-text predicate `contains` supplied by Wais. We give below the corresponding part of the interface:

```
1  <fmodel name="waisfmodel">
2   <fpattern name="Fworks">
3    <node label="works" bind="none" inst="ground">
4     <star inst="none">
5      <value pattern="work" bind="tree"/>
6     </star></node>
7   </fpattern>
```

```
8   </fmodel>
9
10  <operation name="bind" kind="algebra">
11   <input>
12    <value model="Artworks_Structure" pattern="works"/>
13    <filter model="waisfmodel" pattern="Fworks"/>
14   </input>
15   <output>
16    <value model="yat" pattern="Tab"/></output>
17  </operation>
18  <operation name="select" kind="algebra"></operation>
19  <operation name="contains" kind="external">
20   <input>
21    <value model="Artworks_Structure" pattern="Work"/>
22    <leaf label=String /></input>
23   <output><leaf label="Bool"/></output>
24  </operation>
```

Note that, as opposed to the O_2 interface, the *Fpattern* here is very restrictive: it only permits to bind subtrees corresponding to full documents (i.e., only `work` elements). Yet, not much has been achieved since the mediator does not know the semantics of the `contains` predicate, the only one that can be pushed to this source. Hopefully, some connection exists between `contains` and the equality predicate that exists in our algebra. More precisely, a query asking for works by impressionist artists could be evaluated by (i) a full-text search for works containing the string "impressionist" followed by (ii) a standard evaluation of the equality predicate within the mediator. This is expressed with the following equivalence, that we give here in a more readable form than its original in XML:

```
Select($x=$y, Bind(works, works*work[F($x)]))==
Select($x=$y,Select(contains($w,$y),
       Bind(works, works*work($w)[F($x)])))
```

As expected, the equivalence states that starting from a selection with equality over the result of a Bind (`F($x)` denotes here an arbitrary sub-filter with a variable `x`), one can add a more general `contains` predicate over the root of the document (`$w`).

4.3 Related work

In Garlic [24], source capabilities are coded by the programmer within the corresponding wrapper. They remain unknown to the optimizer, that must communicate with the wrappers at optimization/evaluation time to know what part of the query has been accepted and what remains to be processed. In Disco [39, 25], the description of source operations is not typed, which entails extra work for the optimizer in order to match the generated plans against the imported query descriptions. In TSIMMIS [33], optimization opportunities are reduced since the interface language is capable of describing only sets of queries rather than full query languages. To the best of our knowledge, YAT is the only system allowing a generic and complete description of query capabilities for structured sources in such an heterogeneous environment.

5 Optimizing with query capabilities

As pointed out earlier in the paper, optimization techniques from relational and object databases [23, 14] can be applied directly on the corresponding operations in our algebra. In this section, we introduce rewriting techniques for the new *Bind* and *Tree* operators.

5.1 XML queries and Bind rewriting

The *Bind* operation captures some of the most powerful features of XML query languages, like vertical and horizontal navigation, and type filtering. As it is a potentially expensive operation, it is crucial to understand how to simplify and/or rewrite a *Bind*. First, a simpler *Bind* has a better chance to be pushed to a source. Moreover, *Bind* entails navigation that can be costly and should be transformed into more traditional associative access.

Bind and vertical navigation

The upper left part of Figure 7 shows the binding operation over `artifacts`, taken out from the algebraic translation of our view definition (Figure 5). This *Bind* corresponds to a vertical navigation from the set of artifacts down to their local attributes (e.g., `title`) and further down to the information contained in their associated set of owners. Navigation through nested collections is usually captured in object algebras by a join whose right input depends on the left (i.e., *DJoin* in our algebra [14]). Hence, the equivalence between *Bind* and *Djoin* shown in the upper middle part of the figure is not surprising: we can see how *Bind* can be split into more elementary ones, connected through a *DJoin*[2]. As a reward, we can apply classic *DJoin* rewritings and transform navigation into associative access: for instance, in the upper right part of the figure we exploit the `persons` extent to transform the *DJoin* into a standard *Join* supporting more efficient evaluation algorithms.

A complex *Bind* can always be splitted into elementary *Bind*s (i.e., with only one-level deep filters), connected together through *DJoin*s. Another possibility is to split a complex *Bind* into a linear sequence of elementary ones, each one navigating down the result of the previous one. The lower left part of Figure 7 illustrates this rewriting on the *Bind* operation over `artworks` (part of the **Q1** algebraic expression given in Figure 5). Among other things, this rewriting is useful to simplify query compositions or push some evaluation to a source.

Bind, horizontal navigation and type filtering

The absence of type information is usually bad news. Indeed, when a *Bind* operation features a complex filter and no structural information is available, the only

[2] Note the introduction of the new variable `$x` that is removed afterwards by a projection.

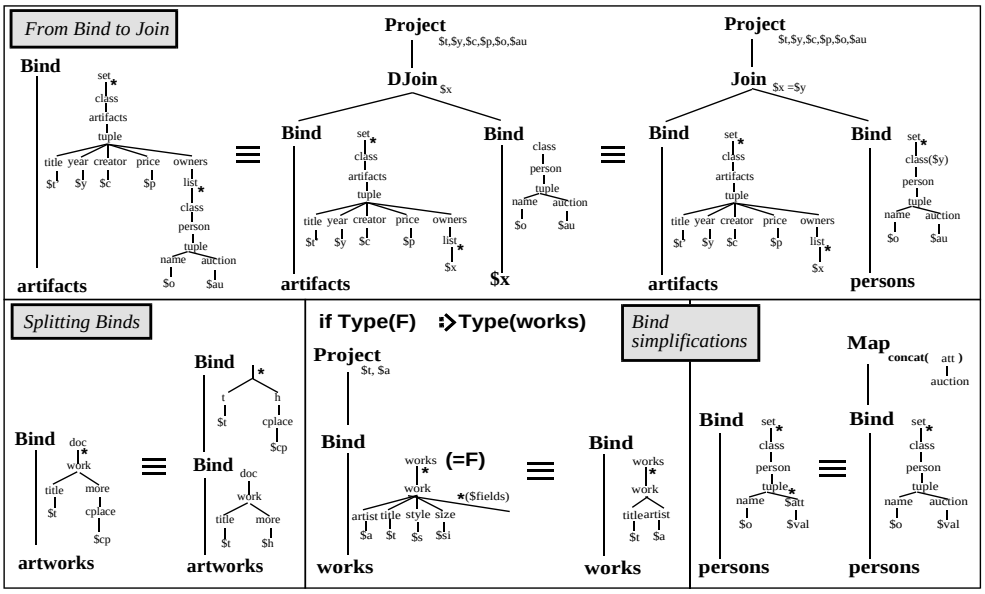

Figure 7: Algebraic Equivalences

evaluation strategy is to navigate through the whole data graph. This is usually what happens in purely semistructured systems. In this case, adding specialized indexes, like in [27], is the only way to achieve reasonable performances. Hopefully, we often have more interesting opportunities, using type information about the data (coming from the source) or the filter (coming from the query). This is particularly useful for queries mixing structured and semistructured data.

Semistructured queries over structured data
By semistructured queries, we mean queries that access both structure and content, e.g., by using tag variables or flexible type filtering. To illustrates this scenario, the lower right part of Figure 7 retrieves the attribute names of `person` objects. Because we have precise type information (see Figure 3), we can simplify the filter, as shown on Figure 7. Note this resembles rewriting techniques for generalized path expressions [12, 20]. This rewriting has several benefits, the most obvious of which being that the *Bind* operation can now be pushed to O_2!

Structured queries over semistructured data
Consider the partially structured XML artworks of our example and assume a user is only interested in the `title` and `artist` elements of artifacts. As illustrated on the lower middle part of Figure 7, this corresponds to a projection that can be used to rewrite the *Bind* operation and simplify the query. Doing so, we must be careful not to change the type filtering semantics of the *Bind*: a sufficient condition for the equivalence to hold is for the type of `works` to be an instance of the type of the filter.

5.2 XML views and Tree-Bind rewriting

The *Tree* operation captures the restructuring semantics of a query or view definition: it features grouping and sorting which are typically expensive operations. A *Tree* can be rewritten as sequence of *Group*, *Sort* and nested *Map* operations, on which existing optimization techniques can be used [14, 10]. Nevertheless, the evaluation of a *Tree* will remain costly if applied on a large amount of data. This is usually not the case with user queries, but may occur when constructing the view. Thus, it is very important to eliminate intermediate *Tree* operations resulting from the composition of queries with the view definition.

It is now time to go back to the evaluation of query **Q1** (see page 4). The left part of Figure 8 presents the algebraic translation of **Q1** composed with the view definition. This complex expression corresponds to a naive evaluation strategy in which the view is materialized, then the query evaluated on the result. Fortunately, our XML algebra comes equipped with all the equivalences we need to rewrite it into the expression on the right part of the figure. Due to space limitations, we only sketch the optimization process here (see [36] for more details).

The first essential step, illustrated by arrows in Figure 8, is to get rid of the *Bind*-*Tree* sequence that appears at the frontier between view definition and query. To do so, we first use *Bind*-Split equivalence given in Figure 7: this introduces an instantiation relationship between the filters of the lower *Bind* and of the *Tree*. Given this relationship, a second equivalence can be used to rewrite the *Bind*-*Tree* sequence in a simple projection with renaming. We are now mostly dealing with operations on which standard rewritings

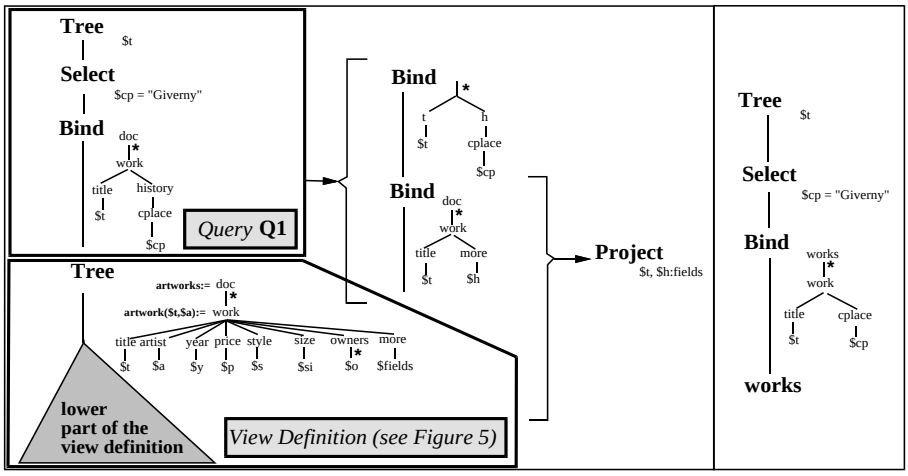

Figure 8: Optimization of Q1

apply. Because all artifacts are available in the XML source, we can push the projection down and: (i) eliminate the branch corresponding to the O_2 source, (ii) simplify the *Bind* on the XML source. Finally, using the *Bind*-Split equivalence in the other way, we can merge the remaining filters to obtain the final expression. Note that we could further optimize the query by using the XML source full text capabilities: this is the subject of the next section.

5.3 Capability-based rewriting

Exploiting source capabilities during query processing is definitely the most important technique in a distributed context. Indeed, pushing some of the query evaluation to an external source allows: to reduce the processing time by using source specific indexes or similar fast access structures; to minimize the communication costs between the sources and the mediator, as well as the conversion costs to the middleware model; to limit the system resources (e.g., memory) required by the mediator; and to benefit from possible parallelism introduced by remote query execution. The next example shows how description of source capabilities from Section 4 can be used during optimization.

Q2: *Which impressionist artworks are sold for less than 200,000.00?*

MAKE *answer [title:$t, artist:$a, price:$p]
MATCH works **WITH** doc *work [title:$t,artist:$a, price: $p,style: $s]
WHERE $p < 200000 **AND** $s = "Impressionist"

The algebraic translation of the query is shown on the left-hand side of Figure 9, along with the equivalence that transforms the *Bind-Tree* sequence into a *Project* operation. The optimized version, shown on the right-hand side, would be evaluated in the following way: first, the XML-Wais source (lower left part) is asked for all artworks containing the string "Impressionist". Next, a second *Bind* is applied to extract the `title`, `artist` and `style` elements from the selected artworks. Then, *for each* pair of title and artist, the O_2 source is called to retrieve the corresponding artifact information. This aspect is due to the *Djoin* operation that corresponds to a nested loop evaluation with values of variables `$t` and `$a` passed from the left-hand side to the right-hand side. Such "information passing" is a classical technique in distributed query optimization [30, 21].

Now, to obtain this plan, the optimizer performs several rounds of rewritings. The first round is quite similar to the one we gave for query **Q1**: after the *Bind-Tree* simplification, the projection is used to simplify the *Bind* on each source and selections are pushed. The goal of the second round of rewritings is to push as much evaluation as possible to the sources. On the O_2 side, little work is required since, as explained in Section 4.1, both *Bind* and selection can be trivially transformed into an OQL query. On the XML-Wais side, the optimizer tries to match the *Bind* operation with the Wais capabilities that have been declared. As, the only possibility is to push a simple *Bind* on XML documents along with a `contains` predicate, the optimizer: (i) introduces a *Select* with `contains` and (ii) splits the *Bind* to match the Wais capabilities description. The first step requires the equivalence declared in Section 4.2, connecting the selection with equality and the selection with `contains`. The second step simply uses the *Bind*-Split equivalence given in Figure 7. Finally, a last round of optimization determines possible information passing between sources and it is based on standard rewritings between *Join*s and *Djoin*s.

6 Summary

We have presented a framework for efficient query evaluation in XML integration systems. It relies on

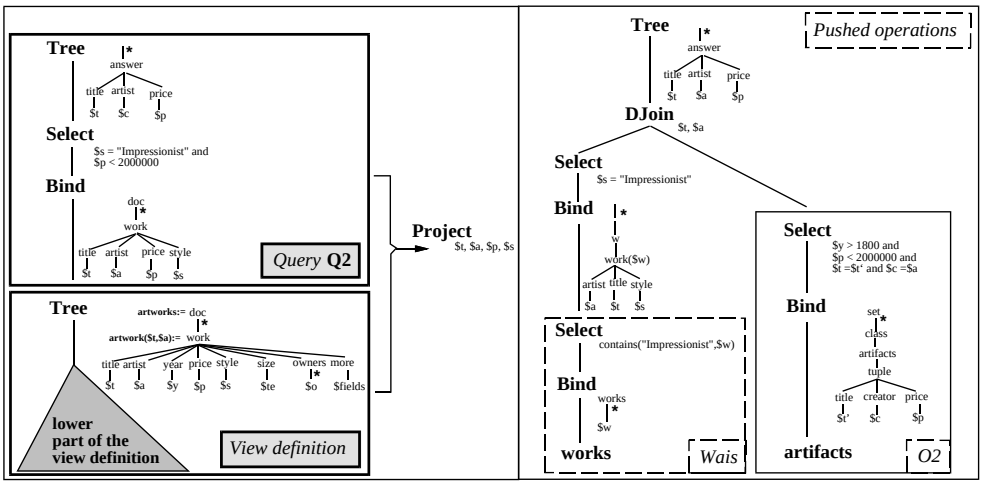

Figure 9: Algebraic translation and optimization of **Q2**

a general purpose XML algebra that captures the expressive power of semistructured or XML query languages and that can be used to wrap structured languages such as OQL or SQL. This algebra comes with equivalences to optimize of query compositions, to exploit type information and to push query evaluation to the external source. This work takes place within the context of the YAT System [36], currently developed at Bell Labs and INRIA[3]. The new XML version of the system, with its algebraic evaluation engine, is running and stable. The implementation of the optimizer is still on-going, based on heuristics and a simple linear search strategy consisting of the three rewriting rounds presented in last section.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, Apr. 1997.

[2] S. Amer-Yahia, S. Cluet, and C. Delobel. Bulk loading techniques for object databases and an application to relational data. In *Proceedings of International Conference on Very Large Databases (VLDB)*, New York, Aug. 1998.

[3] C. K. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 597–599, Philadelphia, Pennsylvania, June 1999. Demonstration.

[4] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of International Conference on Database Theory (ICDT)*, Lecture Notes in Computer Science, Jerusalem, Israel, Jan. 1999.

[5] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *International Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Pennsylvania, June 1999.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, Feb. 1998. `http://www.w3.org/TR/REC-xml/`.

[7] P. Buneman, S. B. Davidson, K. Hart, G. C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 158–169, Zurich, Switzerland, Sept. 1995.

[8] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas II, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach. In *Research Issues in Data Engineering*, pages 124–131, Los Alamitos, California, Mar. 1995.

[9] R. G. Cattell. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[10] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 354–366, Santiago de Chile, Chile, Sept. 1994.

[11] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.

[12] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[13] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 177–188, Seattle, Washington, June 1998.

[3] `http://www-rocq.inria.fr/~simeon/YAT/`

[14] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proceedings of International Workshop on Database Programming Languages*, pages 226–242, New York City, USA, Aug. 1993.

[15] S. Cluet and G. Moerkotte. Query processing in the schemaless and semistructured context. unpublished, 1996.

[16] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to the World Wide Web Consortium, Aug. 1998. `http://www.w3.org/TR/NOTE-xml-ql/`.

[17] M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. Warehousing and incremental evaluation for web site management. In *Proceedings of $14^{ièmes}$ Journées Bases de Données Avancées*, Hammamet, Tunisie, Oct. 1998.

[18] M. F. Fernandez, J. Siméon, D. Suciu, and P. Wadler. A data model and algebra for XML query. Communication to the W3C, Jan. 2000.

[19] M. F. Fernandez, J. Siméon, and P. Wadler (editors). XML query languages: Experiences and exemplars. Communication to the W3C, Sept. 1999.

[20] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, Orlando, Florida, Feb. 1998.

[21] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999. to appear.

[22] G. Gardarin, S. Gannouni, B. Finance, P. Fankhauser, W. Klas, D. Pastre, R. Legoff, and A. Ramfos. IRO-DB : A distributed system federating object and relational databases. In *Object Oriented Multibase Systems : A Solution for Advanced Applications*. Prentice Hall, 1995.

[23] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[24] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 276–285, Athens, Greece, Aug. 1997.

[25] O. Kapitskaia, A. Tomasic, and P. Valduriez. Dealing with discrepancies in wrapper functionality. In *Actes des $15^{ièmes}$ Journées Bases de Données Avancées (BDA'97)*, pages 327–349, Grenoble, France, Sept. 1997.

[26] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous information sources. In *Proceedings of International Conference on Cooperative Information Systems (CoopIS)*, pages 144–156, Brussels, Belgium, June 1996.

[27] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Edinburgh, Scotland, Aug. 1999. to appear.

[28] J. Melton and A. R. Simon. *Understanding the New SQL: A complete Guide*. Morgan Kaufmann, 1993.

[29] A. Michard, V. Christophides, M. Scholl, M. Stapleton, D. Sutcliffe, and A.-M. Vercoustre. The aquarelle resource discovery system. *Computer Networks and ISDN Systems*, 30(13):1185–1200, Aug. 1998.

[30] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[31] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 413–424, Bombay, India, Sept. 1996.

[32] Y. Papakonstantinou, H. Garcia-Molina, and J.Widom. Object exchange across heterogeneous information sources. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 251–260, Taipei, Taiwan, Mar. 1995.

[33] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. D. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings International Conference on Deductive and Object-Oriented Databases (DOOD)*, volume 1013 of *Lecture Notes in Computer Science*, pages 97–107. Springer-Verlag, Singapore, Dec. 1995.

[34] U. Pfeifer. *freeWAIS-sf*. University of Dortmund, 0.5 edition, Oct. 1995.

[35] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). Workshop on XML Query Languages, Dec. 1998. W3C.

[36] J. Siméon. *Intégration de sources de données hétérogènes (Ou comment marier simplicité et efficacité)*. PhD thesis, Université de Paris XI, Jan. 1999.

[37] J. Siméon and S. Cluet. Design issues in XML languages: A unifying perspective. Draft manuscript, Oct. 1999.

[38] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema parts 1: Structures. W3C Working Draft, Sept. 1999.

[39] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 449–457, Hong Kong, May 1996.

[40] L.-L. Yan, M. T. Özsu, and L. Liu. Accessing heterogeneous data through homogenization and integration mediators. In *Proceedings of International Conference on Cooperative Information Systems (CoopIS)*, Charleston, South Carolina, June 1997.

[41] Information retrieval (z39.50): Application service definition and protocol specification. NISO Press, Bethesda, MD, 1995. ANSI/NISO Z39.50-1995.