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 (portals, e-commerce, digital libraries, etc.) require integrated access to various information sources (from traditional RDBMS 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 query evaluation in XML integration systems. 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 [9] is becoming widely used for the development of Web applications that require data integration (e.g., portals, e-commerce, digital libraries). Although fashion surely accounts for some of XML's popularity, it is also justified on technical grounds. Notably, 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 more traditional integration software [47, 31, 21, 11, 49, 33, 27, 10]. 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[1]. For this application, we need to integrate two sources: one, highly structured, is an object database that contains trading information; the other is a partially structured document repository that contains descriptive information about artistic work and is full-text indexed with Wais[2]. Figure 1 shows an XML representation of some sample data from these two sources.

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

[1]Similar to, e.g. www.artdata.com, www.christies.com or www.sothebys.com

[2]`http://ls6-www.informatik.uni-dortmund.de/ir/projects/freeWAIS-sf/`

```
<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

There are four main advantages in using XML for such an application. First, because XML is a very flexible format, it can be used to represent both structured and semistructured information (see Figure 1) from our sources. Second, it is very easy to convert data into XML, and to do so in a generic fashion. In our example, this means that the person in charge of developing the application will not spend too much time generating the XML displayed in Figure 1. Third, there exist many languages allowing declarative integration of XML data (e.g. MSL [39], StruQL [23], YAT$_L$ [18] or XMAS [42]). Finally, being a standard, XML facilitates interoperability. However, query processing in XML-based integration 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 [25]. 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 [9]) is not sufficient to capture rich type systems (e.g., an object database schema) or, conversely, partially structured documents (e.g., in Figure 1, `work`s might come with mandatory elements as well as elements not known in advance, like `history` or `cplace`). Several recent proposals (e.g., XML Schema [7] or DCD [8]) are studying this issue, but no definitive standard is available yet. In [18], 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 [41] are used to describe source capabilities. However, this does not allow an exhaustive description of a source capabilities (e.g., all possible queries on the example object database) and implies costly *ad hoc* development (i.e. to code, for each application, the translation of a given list of queries).

- *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 to be able to exploit partial type information and very heterogeneous source capabilities.

In this paper, we propose an algebraic framework and optimization techniques to address the last two of the above 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 languages with their respective specificities.

A source description language. We show how this algebra can be used to wrap full text queries or structured query languages such as OQL or SQL in a complete (i.e. allowing exploitation of sources full query capabilities) 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.

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. In Section 5 we present the optimization techniques and the system current implementation status. We conclude in Section 6.

2 XML Integration with YAT

The YAT System is a semistructured data conversion system [18, 44] that we are currently turning in to a full-fledged XML integration system. As we already presented in [45], 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:

1. `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 (here the `art` database) as well as the system query capabilities (i.e., it wraps OQL, we will see how in Section 4).

2. `christop` wraps the cultural source with another generic wrapper. The `xmlwais` wrapper understands XML data, typed with our type system and indexed by Wais. It expects a standard Wais source configuration file (`museum.src` in the example) as parameter.

3. `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 (`view1.yat`).

```
--------------------------------------------------------------------------------
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";
yat>
```

Figure 2: Installing Wrappers and Mediators

Before we take a closer look at the integration program itself, let us first give the structural information, as exported by the two wrappers. Note that for interoperability reasons, wrappers and mediators communicate data, structures and operations in an XML syntax (see [16] for the complete XML interfaces).

The YAT type system is based on a simple yet powerful mechanism that allows to represent information at various levels of genericity (model, schema, data) and to understand the connection existing between these levels. As will be explained in Section 4, we rely on this specific feature to wrap query languages. We present it here briefly (see [18] for a complete description).

Figure 3 gives a graphical representation of the YAT data model and of the type information exported by our two wrappers. The left hand-side of the figure shows the O_2 data model (on top) and the schema of the `art` database (in the lower part). Note that (i) bold fonts denote identifiers, (ii) the `&` symbol denotes references to identifiers, (iii) the $\star$ and $\vee$ symbols are used to denote respectively multiple occurrences and alternatives. Thus, an O_2 type is described as being either an atomic type, or a tuple, or a collection or a reference to a class. A tuple type is represented as a collection of linear subtrees, each associating a symbol to a value of some type. One interesting property of this representation is that the `Artifact` schema is recognized by the system as an instance of the O_2 model (which is denoted `Artifact <: ODMG`).

Let us now take a look at the representation of the documents exported by the `xmlwais` wrapper in the lower left part of the figure. Each is described as a sequence of mandatory elements, followed by a collection of optional and unknown elements (called `Fields`). Note the flexibility of this model that can capture partially structured information, avoiding the dilemma of choosing either one of

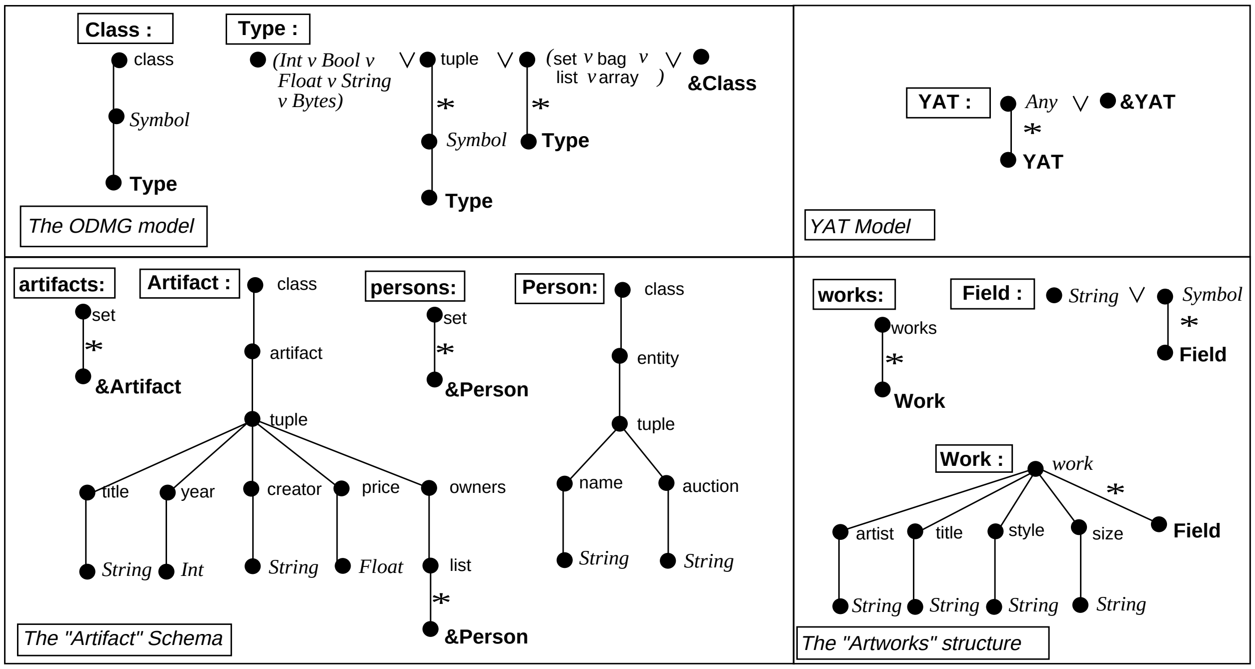

Figure 3: O_2, XML-Wais and YAT mediator structural metadata

the valid (i.e. XML with precise schema) or well-formed (XML with no schema at all) strategy.

Finally, on the top right part of Figure 3, one sees a representation of the YAT meta-model itself, that captures any tree. Once again, the O_2 model, the `Artifacts` schema, and the `Artworks` structure are instances of this almighty model (in fact, we have `Artifact <: ODMG <: YAT`).

Integration programs in declarative languages are usually composed of a sequence of rules or queries [39, 23, 18], whose partial results are connected together through Skolem functions. Figure 2 shows a YAT_L query[3], extracted from program `view1.yat`, that constructs a document (`artworks`) in the integrated view. This document contains both the trading and descriptive information about each work of art which is available in the two sources.

The query consists of three clauses. The **MATCH** clause performs pattern-matching: filters are used to navigate in the source data and bind variables to the appropriate information (e.g., the artifact title to variable `$t`, the list of optional XML elements to `$fields`). YAT_L filtering mechanism relies on type instantiation: if a tree is an instance of a filter, then one can deduce a mapping between values and variables. Otherwise, a type error occurs. The **WHERE** clause fulfills the usual function. The **MAKE** clause constructs the result by creating a new tree 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 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 the consistency of an application or signal source modifications.

[3]In its new syntax [24, 46].

```
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'
```

Figure 4: Integrating information about the works of Art

2.1 Technical challenges in query processing

The above example illustrates the simplicity of XML-based integration. Apart from the quality of structural descriptions provided by YAT, other semistructured or XML integration systems (e.g. TSIMMIS [40], MIX [4]) offer similar functionalities. But we still have to evaluate user queries in an efficient way. As an invitation to proceed further, assume that a user, after having noticed that some artworks had a creation place (`cplace` field), 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: (i) how to compose it with the view definition (note that **Q1** accesses the semistructured fields of artwork documents), (ii) how to understand that only the XML-Wais source is needed to answer the query, (iii) how to exploit the Wais textual queries to avoid downloading all documents. In the following, we demonstrate that the algebraic framework we propose can answer successfully all these questions.

3 The YAT operational model and XML Algebra

Choosing the good operational model for information integration is a strategic decision. As we will see in the remainder of the paper, it is the main tool for both the generic description of source query capabilities and the XML query optimization. We adopt a simple but expressive operational model: it relies on a functional approach allowing arbitrary compositions of any side-effect free functions[4] as well as function calls, and provides a fixed set of predefined operations - the so-called YAT XML algebra. This algebra has been designed with respect to the following requirements:

Expressive power. The algebra must capture evaluation of query and integration languages, along with their XML-specific features. Notably it must support complex pattern matching primitives including ordered navigation (like in XQL [43] or YAT_L), different kinds of variables (atomic values, whole sub-trees, tag or index variables) as well as Skolem functions.

Support for flexible typing. XML favors flexibility and most XML query languages are not typed. Yet, we also want to capture the properties of structured languages. Thus, the algebra should support both flexible type filtering (in the style of Lorel[1], XML-QL[22]) and more strict forms of typing (for languages such as OQL [17, 12]).

Support for optimization. Of course, the algebra should come equipped with a number of equivalences offering interesting optimization opportunities.

The algebra is an extension of the object algebra of [19]. In this section, we present the operators that we introduced to deal with tree structures and recall briefly the others. Next, we show how queries are translated in our operational model. Finally, we give a brief overview of alternative proposals.

[4]With the exception of Skolem functions that feature a limited form of side-effect.

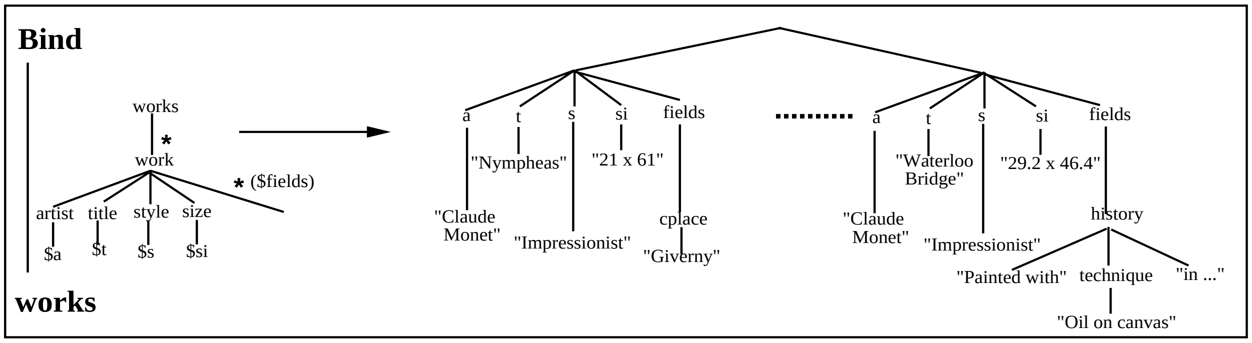

Figure 5: A Bind operation and resulting Tab structure

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. Then, on these *Tab* structures we can apply standard operators such as *Join*, *Select*, *Project*, etc. Finally, an inverse operation to *Bind*, called *Tree*, generates a new nested XML structure.

The Bind operator extracts data from some arbitrary input tree according to a given filter (i.e. a tree featuring distinct variables) and produces a tabular representation of the variable bindings resulting from the filtering operation.

Figure 5 illustrates this mechanism with an example. The *Bind* operation is applied on the tree representing the XML collection of works with a filter asking for a bind of each work title(`$t`), artist (`$a`), style (`$s`), size (`$si`) and optional elements (by putting the variable `$fields` on the edge, we require the construction of a subtree with all the branches that are not bound by the previous variables). Note the similarity between the *Tab* structure and a ¬1NF relation.

Bind is quite a powerful operation, providing support for type filtering and navigation, both vertical and horizontal (through horizontal regular expressions - see `$fields` bound to the remaining subtree). However, as we will see in Section 5.1, it can be decomposed into more simple operations when necessary. More details about *Bind* filter patterns can be found in [18], while their extension to allow flexible type filtering is presented in [46].

The Tree operator is applied on *Tab* structures and returns a collection of trees conforming to some input pattern. Figure 6 illustrates its use. Here, the *Tree* operation is applied on the result of the previous *Bind* (`F[$t,$a,$s,$si,$fields]` denotes the filter of Figure 5). It groups the works according to their artist (`*($a)`) and creates a new identifier for each artist (`artist($a)`). The trees associated to the artist identifiers regroup the title of their work (`$t` after the `*`). Note that this is somehow equivalent to a grouping operation.

Skolem functions are used to create new identifiers and perform value assignment. In our algebra, Skolem functions do not create values but have side effect on the integrated view. They are somehow orthogonal to the rest of the algebra.

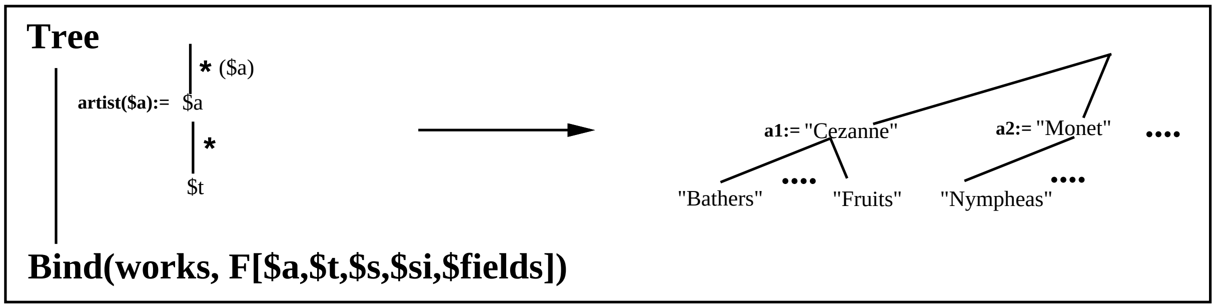

Figure 6: The Tree operation

The other operators of the algebra are those of the object algebra of [19]. *Select*, *Project*, *Join*, *Union*, *Intersection* come from relational. Classic object operations are: *Group*, *Sort*, *Map* and *D-Join* (for dependency join) which is used for navigating within nested collections. Their definition on *Tab* structures rather than collections of tuples is straightforward[5]. For lack of space, 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* or *Map* has to be applied.

As most of the algebra is composed of standard operators, we can take advantage of their well-known properties and reuse all the equivalences from object-oriented optimization (e.g. covering standard relational ones or those for nested queries [19]). Concerning the distinctive characteristics of XML data, note that:

- *Bind* and *Tree* are two frontier operations isolating processing specific to trees from more standard one.
- By allowing recursive calls in the algebra (which was not the case in [19]), we capture generalized path expressions [14, 1] (See [46] for more details). In [15], we studied their optimization. We do not address this issue again here (also see [25] for more on this issue).

Finally, one important characteristic of this algebra is that it is independent of any underlying physical access structure. Therefore, it 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 whether they are virtual (e.g. accessed through wrappers as in our context). Indeed, in Section 5 we will present useful rewritings for both cases.

3.2 Expressive power and YAT$_L$ algebraic translation

Our algebra offers the arsenal necessary to express the evaluation of existing XML query languages (notably, YAT$_L$, Lorel, XML-QL and XQL). Figure 7 shows the algebraic translation of the view definition of Figure 2 and of query **Q1** (translation of other XML query languages would be performed in a similar manner[6]). It has been obtained using the following translation steps:

[5]Although not illustrated here, [19]'s algebra is multi-sorted, which corresponds to Tab's with collections of different kinds: set, list or bag.

[6]Note that the translation of some particular features, for instance preservation of deeply nested structure with order in XQL, would be more involved.

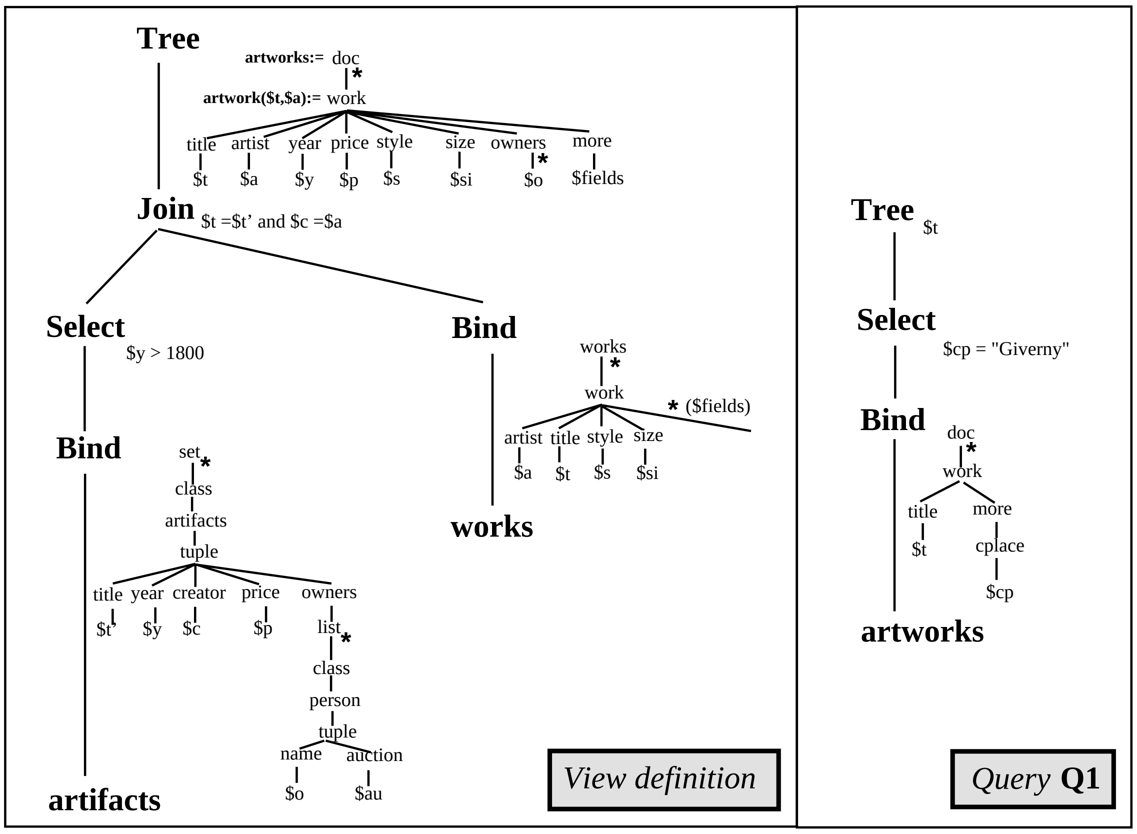

Figure 7: Algebraization of YAT$_L$ queries

1. Named documents (e.g. `artifacts`) are the input operations of the algebraic expression.
2. Each statement of the **MATCH** clause is translated into a *Bind* operation that captures its filtering/binding semantics, and creates a *Tab* structure suitable for further processing in the algebra.
3. The connection between the various inputs is materialized using a *Join* operation.
4. The other predicates in the **WHERE** clause are translated into a *Select* operation.
5. Finally, the **MAKE** clause is translated using the *Tree* operation.

3.3 Related work

Several algebras for semistructured data have been proposed. The algebra presented in [34] is a physical algebra, aimed at the optimization involving Lore indexes, and does not provide the appropriate expressive power (notably horizontal navigation, Skolem functions, type filtering). A physical algebra is also used in XML-QL query engine. An XML algebra is proposed in [5] to support XML queries, but it does not provide the appropriate expressive power and optimization properties are unclear (e.g. the Join operation is not commutative anymore). Finally, SAL [6] is a logical XML algebra, but it does not support horizontal navigation, sorting, Skolem functions and type filtering.

Compared to object algebras, YAT *Bind* resembles the *Scan* operator of [20] (minus the condition, plus potentially complex patterns). An object algebra with side-effects similar to that of Skolem functions has been presented in [2].

4 Generic wrapping of sources query capabilities

As we explained in Section 2, each wrapper exports its source query capabilities. In this section, we explain how wrappers communicate this information to the mediator. More important, we show how the combination of our operational model and type system allows to do this at different genericity levels: from full query languages (e.g., OQL on the ODMG model) to sets of queries (e.g., a method on an O_2 schema or a more exotic textual predicate on XML elements).

Wrapping source operations in YAT is performed in two steps that concern (i) signature and (ii) semantics. The first step is in fact the most essential. The second is needed only for special optimization purposes as will be explained later. In most cases, both steps are performed automatically by the wrappers. However, for some exotic sources (i.e., those featuring operation that cannot be captured by the core operational model), the second step must be performed manually. As an example of the first step, 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:

```
<operation name="external">
  <operation name="current_price">
    <input>
      <value model="Artifact_Schema" pattern="Artifact"/></input>
    <output>
      <leaf label=Float /></output>
  </operation>
</operation>
```

Mainly, one needs to note the `input` and `output` elements: `current_price` is said to take an `Artifact` and to return a float. Note that this declaration is performed automatically by the the O_2 wrapper by simply querying the O_2 schema manager. Once the method is wrapped, it is available to the integration programmers, and, potentially to the end users (if the programmer decides so).

Let us now see how OQL and Wais are captured in this manner before looking at some related work. Optimization issues are discussed in Section 5.

4.1 Describing OQL capabilities

We consider here the description of OQL [12, 17]. Obviously, SQL [35, 36] could be described in a similar manner (although the wrapper implementation is more complicated due to the non-functional nature of SQL).

YAT operational model borrows a large part of OQL algebra [19]. However, YAT supports also two new semistructured operators as well as a different type system. As a matter of fact, whereas YAT captures OQL, the opposite is not true for mainly one reason: OQL binding capabilities are more restricted (e.g., we cannot query schema information). We now explain how we can describe precisely OQL query capabilities taking this restriction into account.

Capturing Binding Capabilities. A *Bind* operation has two parameters: a filter and the data that has to be filtered/bound. In order to distinguish between *Bind* operations that can be actually evaluated by OQL and those that cannot, we first need to specify which are the acceptable filters for OQL. Such a specification of valid filters (called *Fpattern*) is shown on Figure 8. An *Fpattern* is essentially a serialization of the structured patterns illustrated in Figure 3, annotated with different kinds of flags. For instance, in the filter for O_2 classes (pattern with name `Fclass`, line 4) the attributes `bind` and `ground` of pattern nodes are used to declare that (i) only subtrees corresponding to actual O_2 objects or values can be bound (`bind="tree"`, line 5) (ii) extraction of class schema information is prevented (`bind="none"`, line 6) and (iii) the name of classes in a schema specific filter has to be instantiated (`inst="ground"`, line 6).

The good thing about this *Fpattern* description is that the integration programmer does not need to see it. It is coded by the YAT developers and embedded within the O_2 wrapper. This also applies to the XML descriptions given next.

Description for OQL. Below is a subset of the operational interface of the O_2 wrapper:

```
1    <omodel name="o2omodel">
2      <operation name="algebraic">
3        <union>
4          <operation name="bind">
5            <input>
6              <value model="o2model" pattern="Type"/>
7              <filter model="o2fmodel" pattern="Ftype"/>
8            </input>
9            <output>
10             <value model="yatstruc" pattern="Tab"/>
11           </output>
12         </operation>
13         <operation name="select"></operation>
14         <operation name="map"></operation>
15         ...
16       </union>
17     </operation>
18
19     <operation name="boolean">
20       <union>
21         <operation name="shalow_eq"></operation>
22         <operation name="leq"></operation>
23         ...
24       </union>
25     </operation>
26     ...
27   </omodel>
```

Note that two kinds of operations are declared: algebraic (line 2) and boolean (line 19) operations. In fact there are others: external (e.g., the *current_price* method), arithmetic (e.g., +),

```
<interface name="o2artifact">
 <operat>
    <fmodel name="o2fmodel">
      <fpattern name="Fclass">
         <node label="class" bind="tree">
            <node label="Symbol" bind="none" inst="ground">
               <value pattern="Ftype"/></node></node>
      </fpattern>

      <fpattern name="Ftype">
        <union>
          <leaf label="Bool"/>
          <leaf label="Char"/>
          <leaf label="Int"/>
          <leaf label="Float"/>
          <leaf label="String"/>
          <node label="tuple" col="set" bind="tree">
             <star inst="ground">
               <node label="Symbol" bind="none"><value label="Ftype"/></node>
             </star></node>
          <node label="set" col="set" bind="tree">
             <star inst="none"><value label="Ftype"/></star></node>
          <node label="bag" col="bag" bind="tree">
             <star inst="none"><value label="Ftype"/></star></node>
          <node label="list" bind="tree">
             <star inst="none"><value label="Ftype"/></star></node>
          <node label="array" bind="tree">
             <star inst="none"><value label="Ftype"/></star></node>
          <ref pattern="Fclass"/>
        </union>
     </fpattern>
   </fmodel>
 </operat>
</interface>
```

Figure 8: O_2 Filter patterns exported in XML

etc. The first algebraic operation we declare is the *Bind* operation. Note that its signature has been specialized using the *Fpattern* `Ftype` introduced in the previous subsection. Other algebraic operators follow, none of which with a specialized signature (e.g., select, map, etc.). To understand why this specialization is not required, we need to remind that our goal is to be able to push operations on the connected sources. An operation can be pushed only on data imported by the source or on the result of a pushed operation. Since *Bind* is always the first operation in a query, we are sure that other pushable algebraic operations will be applied on ODMG compliant data (a Tab captures any o2 collection of tuples) and there is no need to overload the system with unnecessary information. Furthermore, in order to be pushed, all the arguments of an operation must be pushable. For instance, selection and map operations will be pushed with predicates (e.g., `=`, `<=`, etc.)) or functions (e.g., the method `current_price`) understood by O_2. Going back to our integration example, the sequence of *Bind* and *Select* operations illustrated in Figure 7 can be pushed to the O_2 source and it will translated by the wrapper into the following equivalent OQL query:

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

4.2 Describing Wais capabilities

For database people, the most basic operation one can perform on a source is to ask for an entry point (e.g., a relation, a set of objects). However, this seemingly simple operation is not supported by some sources. For instance, many Web sites are only accessible through form-based query interfaces and does not allow to export source's full content. Thus, it is capital to understand the operations supported by these sources, even if not supported by the original YAT model.

Another apparently simple assumption in the database community is that you may view what you query. Again, this is not always true. For instance, the Z39.50 [3] protocol (underlying the Wais retrieval engine and which is widely used for digital libraries) is based on attribute/value textual queries and establish 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 ones [37]. Although we do not illustrate this here, the extensibility of the operational model allows to capture this peculiar feature in a very simple way (by adding new predicates for each queried field and exporting them to the view).

In the sequel, we focus on the wrapping of the full-text capabilities of our XML-Wais source (or more generally of distributed information retrieval protocols and search engines) as well as the declaration of source-supplied equivalences.

Importing the query capabilities of an XML-Wais source. In order to wrap the query capability of the XML-Wais source we need to (i) specify the source *Fpattern*s, (ii) declare that the source supports *Bind* and *Select* and (iii) describe the full-text predicate `contains` supplied by Wais. Note that the *Fpattern* is very restrictive as it only permits to bind subtrees corresponding to full documents (i.e., `work` but not `artist` or any other element). We give below the interface corresponding to the XML-Wais wrapper:

```
 <interface name="xmlartist">
  <operat>
      <fmodel name="waisfmodel">
         <fpattern name="Fworks">
             <node label="works" bind="none" inst="ground">
                 <star inst="none">
                    <value pattern="work" bind="tree"/></star></node>
         </fpattern>
     </fmodel>

     <omodel name="waisomodel">
       <operation name="algebraic">
        <union>
         <operation name="bind">
           <input>
             <value model="Artworks_Structure" pattern="works"/>
             <filter model="waisfmodel" pattern="Fworks"/></input>
           <output>
             <value model="yatstruc" pattern="Tab"/></output>
         </operation>

         <operation name="select"></operation>
        </union>
       </operation>

       <operation name="external">
         <operation name="contains">
           <input>
             <value model="Artworks_Structure" pattern="Work"/>
             <leaf label=String /></input>
           <output>
             <leaf label=Bool /></output>
         </operation>
      </operation>
     </omodel>
  </operat>
```

Once this is done, not much has been achieved since the mediator does not know the semantics of the only predicate that can be pushed to the source. Yet, there exists some connection between full-text and equality predicates. For instance, 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 in our interface language with the following equivalence given, for readability, in a textual form:

```
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 [47, 41, 29], 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 [30], 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 [32, 48], optimization opportunities are reduced since the interface language is capable to describe only sets of queries rather than full query languages. To the best of our knowledge, YAT is the only system allowing generic and complete description of capabilities for arbitrary sources.

5 Exploiting Source Capabilities and Optimization

Our algebra is an object algebra extended to manipulate XML data with two operators (*Bind* and *Tree*). As a consequence, a large number of optimization techniques proposed for the relational or object models [28, 19] are directly applicable. In this section, we introduce rewriting techniques for the new *Bind* and *Tree* operators. Our goal is to optimize arbitrary compositions of user queries with integration views, either locally or by pushing queries to the external sources.

5.1 XML processing and Bind rewriting

The *Bind* operation is used to bind variables, but also to capture some of the most powerful features of XML query languages, notably vertical and horizontal navigation as well as type filtering. *Bind* is a potentially expensive operation. Understanding how to simplify and/or rewrite it is crucial since: (i) a simpler *Bind* has a better chance to be pushed to a source, (ii) *Bind* entails navigation that can be costly and should be transformed into more traditional associative access as much as possible.

Bind and vertical navigation

The left-hand side of Figure 9 shows the binding operation over `artifacts`, taken out from the algebraic translation of our view definition (Figure 7). 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 (*DJoin* in our algebra [19]). Hence, the equivalence between *Bind* and *Djoin* shown in Figure 9 is not surprising: in the middle part of the figure we see that the *Bind* has been split into more elementary ones connected through a *DJoin*. (Note the introduction of the new variable `$x` that is removed afterwards by a projection.) As a reward, we can apply classic *DJoin* rewritings and transform navigation into associative access: for instance, the right-hand side of Figure 9 exploits the `persons` extent to transform *DJoin* into a standard *Join* (for which more efficient algorithms are available). A complex *Bind* can always be splitted into elementary *Bind*s (i.e. with only one-level deep filters), connected together through *DJoin*s.

Instead of using *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 left-hand side

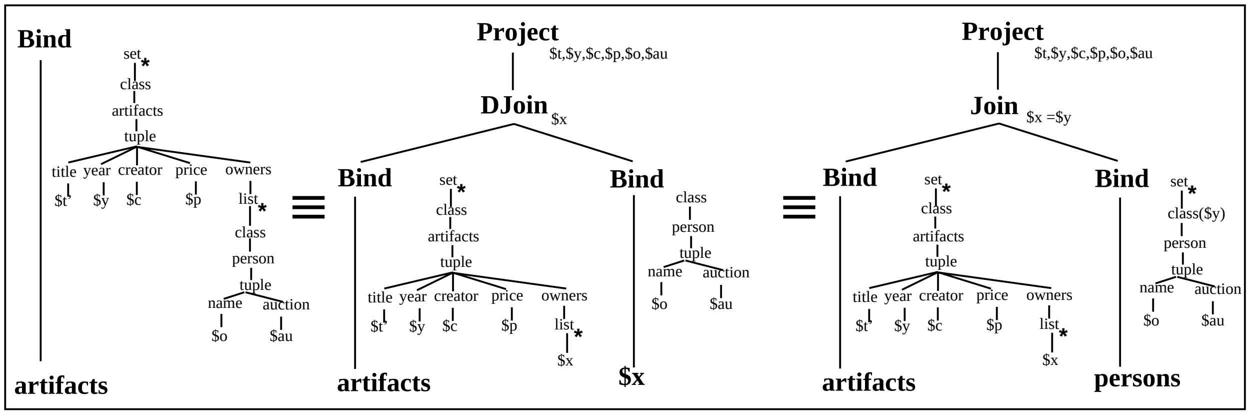

Figure 9: From Bind to Join

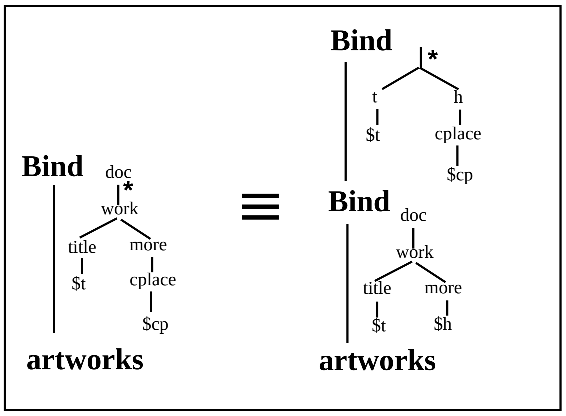

Figure 10: Splitting Binds

of Figure 10 illustrates this by rewriting the *Bind* operation over `artworks` that is part of the **Q1** algebraic expression given in Figure 7. Among other things, this rewriting is useful to, e.g., to simplify query compositions with XML integration views.

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 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 [34], 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 (i.e. coming from the query). This is particularly useful for XML queries mixing structured and semistructured data and it is illustrated next.

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. For instance, the left part of Figure 11 shows how to retrieve the attribute names of `person`

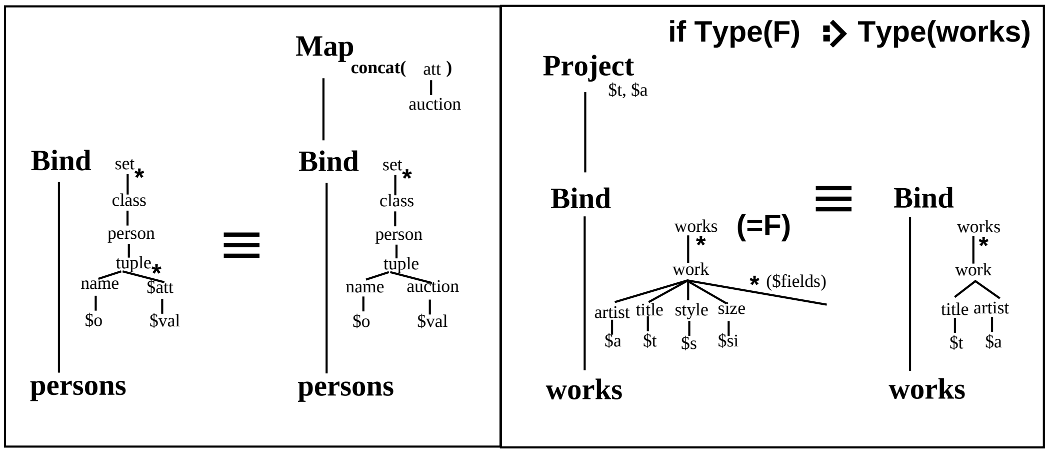

Figure 11: Bind and Map or Project

objects. Fortunately, we have precise type information(see Figure 3) and we can simplify the filter[7], as shown on Figure 11. This rewriting has several benefits, the most obvious of which is that the new *Bind* operation can be pushed to the O_2 source!

Structured queries over semistructured data. Consider for instance the partially structured XML artworks of our example and assume that a user is only interested in the artifacts `title` and `artist` elements. As illustrated on the right side of Figure 11, this corresponds to a projection. By using this projection to rewrite the *Bind* operation, we can 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 to verify, as it is our case, that the type of `works` is an instance of the filter.

5.2 Query composition and Tree-Bind rewriting

The *Tree* operation captures the restructuring semantics of a query or view definition: it features implicit grouping and sorting which are typically expensive operations. As a matter of fact, a *Tree* can be rewritten as sequence of *Group*, *Sort* and nested *Map* operations, on which optimization techniques, e.g. from [19, 13], can be used. However, the evaluation of a *Tree* remains costly if applied on large amounts of data. This is usually not the case with user queries, but may occur when constructing a view. It is therefore particularly important to eliminate the intermediate *Tree* operations resulting from the composition of queries and view definition.

We now go back to the evaluation of query **Q1** (see page 7). The left part of Figure 12 presents the composition of the algebraic translation of **Q1** with the view definition: it is a rather complex algebraic expression. A naive evaluation strategy would first materialize the view, and then evaluate the query. Fortunately our XML algebra comes equipped with all the equivalences we need in order to rewrite it as shown in the right part of Figure 12. Because of space limitations, we only sketch the optimization process here, more details can be found in [44].

[7]Note that it is similar to rewriting techniques for generalized path expressions [15, 25].

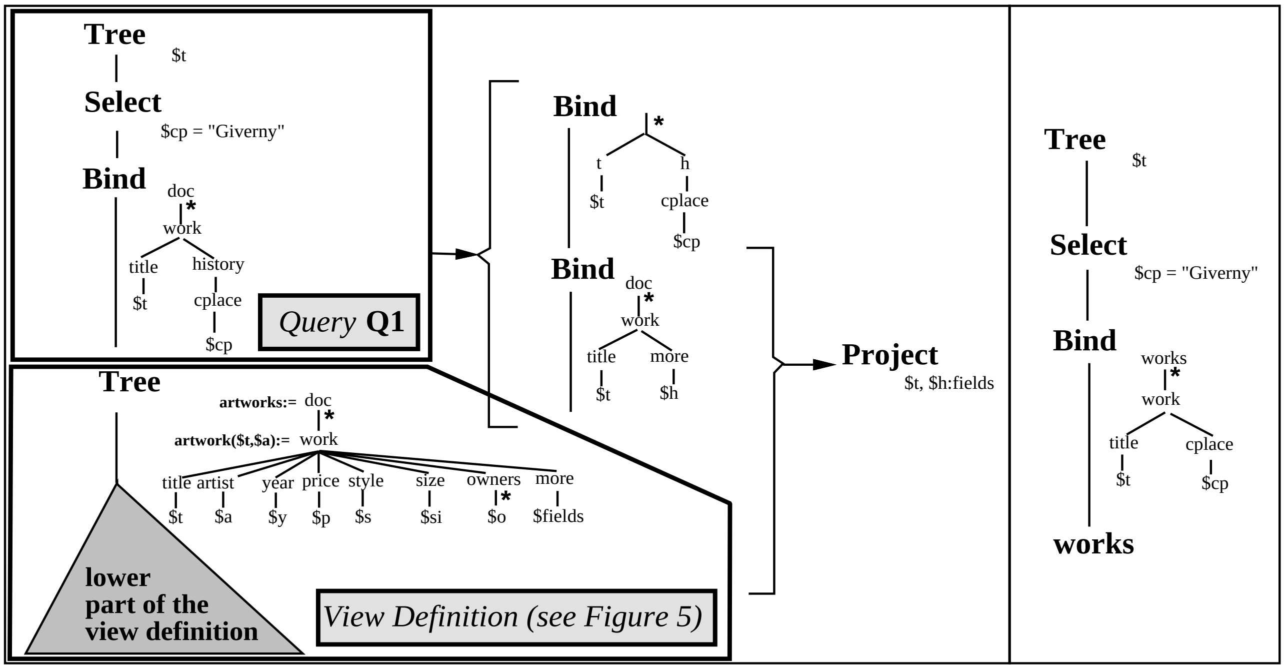

Figure 12: Optimization of Q1

The first essential step, illustrated by arrows in Figure 12, 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 the equivalence from Figure 10 to split the *Bind* operation into two: this introduces an instantiation relationship between the filters of the lower *Bind* and of the *Tree*. Given this relationship, a second equivalence is used to transform the *Bind*-*Tree* sequence into a simple projection with renaming. Once this is done, we are mostly dealing with object operations on which standard rewritings 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 and (ii) simplify the *Bind* on the XML source. Finally, using once more the equivalence from Figure 10 but in another direction, we merge the remaining *Bind* filters to obtain the final expression, on the right side of Figure 12. Note that we could optimize the query further by making use of the XML source full text capabilities. This type of optimization is illustrated in the sequel with another example.

5.3 Source 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 the optimization process.

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 13, along with the equivalence that transforms the *Bind-Tree* sequence into a *Project* operation. The optimized version is illustrated on the right-hand side. Before we describe how it has been obtained, let us see how it can be evaluated: 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 last part is captured by the *Djoin* operation: it corresponds here to a nested loop evaluation with variables `$t` and `$a` passing from one side to the other. This kind of information passing is standard in distributed query optimization [38, 26].

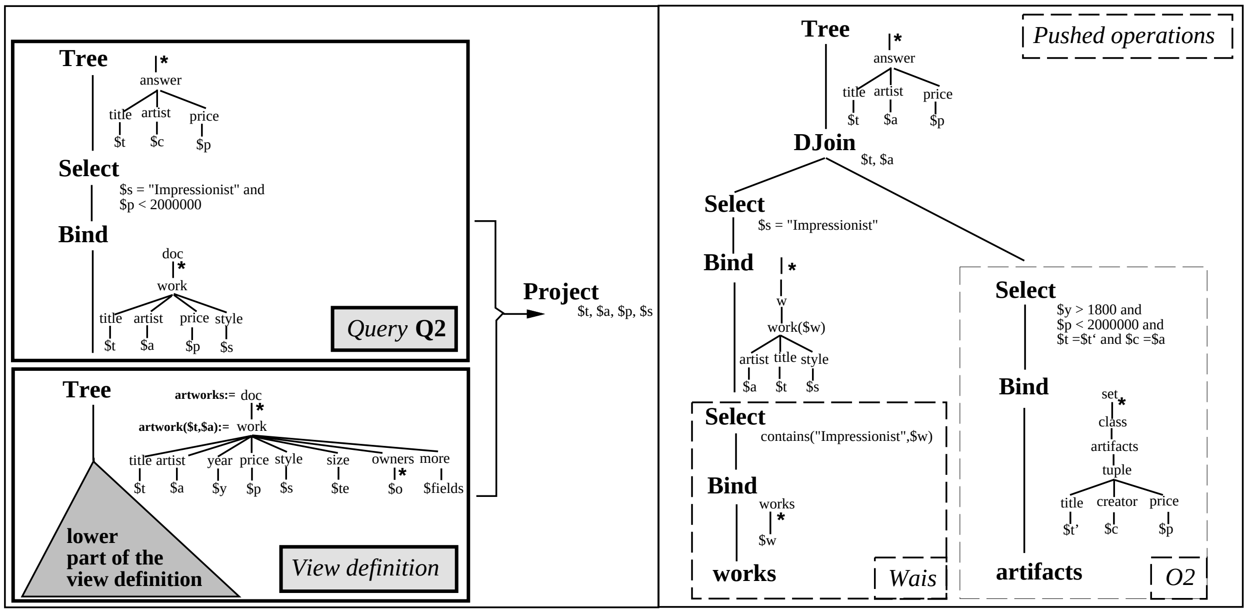

Figure 13: Algebraic translation and optimization of **Q2**

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 equivalence from Figure 10. 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.

5.4 System status

This work takes place within the context of the YAT System [44], currently developed at Bell Labs and INRIA[8]. At the time we write, 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. This first implementation is based on heuristics and a simple linear search strategy consisting of the three rewriting rounds presented previously.

6 Conclusion

We have presented an algebraic framework to support efficient query evaluation in XML integration systems. It relies on a general purpose algebra allowing to capture the expressive power of semistructured or XML query languages but also to wrap, with appropriate type information, more structured query languages such as OQL or SQL. The proposed XML algebra comes equipped with a number of equivalences offering interesting optimization opportunities. Notably, they enable to optimize query compositions, exploit type information and push query evaluation to the external source.

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, April 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, August 1998.

[3] ANSI/NISO. Z39.50-1995 (Versions 2 and 3) Information Retrieval: Application Service Definition and Protocol Specification, 1995.

[4] 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 540–543, Philadelphia, Pennsylvania, June 1999.

[5] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for xml. Communication to the W3C, September 1999.

[6] 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.

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

[7] D. Bleech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. S. Thompson. Xml schema (parts 1 & 2). W3C Working Draft, September 1999.

[8] T. Bray, C. Frankston, and A. Malhotra. Document content description for xml. Submission to the World Wide Web Consortium, July 1998.

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

[10] P. Buneman, S. B. Davidson, K. Hart, 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, September 1995.

[11] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, 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, March 1995.

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

[13] 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, September 1994.

[14] 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.

[15] 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.

[16] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration (full version).
`http://www-db.research.bell-labs.com/user/simeon/yamg_full.ps`, October 1999.

[17] S. Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5):279–305, 1998.

[18] 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.

[19] 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, August 1993.

[20] S. Cluet and G. Moerkotte. Query Processing in the Schemaless and Semistructured Context. unpublished, 1996.

[21] Some commercial integration systems.
http://www.software.ibm.com/data/datajoiner/,
http://www.sybase.com/products/entcon/,
http://www.oracle.com/products/servers/rdb/html/fs_dbi.html,
http://www.unisql.com/product_info/unisqlm.html,
http://www.cincom.com/totalframework/index.html,
http://www.enterworks.com/products.html,
http://www.cerfnet.com/~margie/dii/,
http://www.junglee.com/,
http://www.crossaccess.com/product.htm,
http://www.ibi.com/.

[22] A. Deutsch, Mary Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. XML-QL: A Query Language for XML. Submission to the World Wide Web Consortium, August 1998. http://www.w3.org/TR/NOTE-xml-ql/.

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

[24] M. Fernandez, J.Siméon, and P.Wadler (editors). XML Query Languages: Experiences and Exemplars. Communication to the W3C, September 1999.

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

[26] D. Florescu, A. 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.

[27] 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.

[28] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[29] 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, August 1997.

[30] O. Kapitskaia, A. Tomasic, and P. Valduriez. Dealing with discrepancies in wrapper functionality. In *Actes des 13^{ièmes} Journées Bases de Données Avancées (BDA'97)*, pages 327–349, Grenoble, France, September 1997.

[31] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 251–262, Bombay, India, September 1996.

[32] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, and J. Ullman. Capability-Based Mediation in TSIMMIS. In *Exhibits Program of SIGMOD'98*, pages 564–566, Seattle, Washington, June 1998.

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

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

[35] J. Melton. *ISO-ANSI SQL2*. International Standard Organization and American National Standards Institute, 1988.

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

[37] 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, August 1998.

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

[39] 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, September 1996.

[40] 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, March 1995.

[41] 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, December 1995.

[42] Y. Papakonstantinou and P. Velikhov. Enhancing Semistructured Data Mediators with Document Type Definitions. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 136–145, Sydney, Australia, March 1999.

[43] J. Robie, J. Lapp, and D. Schach. Xml query language (xql). Workshop on XML Query Languages, December 1998. W3C.

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

[45] J. Siméon and S. Cluet. Using YAT to Build a Web Server. In *International Workshop on the Web and Databases (WebDB'98)*, Valencia, Spain, March 1998.

[46] J. Siméon and S. Cluet. Design Issues in XML Languages: A Unifying Perspective. Draft manuscript, October 1999.

[47] 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.

[48] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 256–265, Athens, Greece, August 1997.

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