Querying XML sources using an Ontology-based Mediator

Bernd Amann[1], Catriel Beeri[⋆2], Irini Fundulaki[1], and Michel Scholl[1]

[1] Cedric-CNAM Paris and INRIA-Futurs, France
amann@cnam.fr, fundulak@cnam.fr, scholl@cnam.fr
[2] The Hebrew University, Jerusalem, Israel
beeri@cs.huji.ac.il

Abstract. In this paper we propose a mediator architecture for the querying and integration of Web-accessible XML data sources. Our contributions are (i) the definition of a simple but expressive mapping language, following the *local as view* approach and describing XML resources as local views of some global schema, and (ii) efficient algorithms for rewriting user queries according to existing source descriptions. The approach has been validated by the ST_YX prototype.

1 Introduction

During the last decade, there has been a significant focus on data integration. In a nutshell, data integration can be described as follows: given *heterogeneous* and *autonomous* information sources in a *specific domain of interest*, the goal is to enable users to *query* the data as if it resides in a *single source*, with a *single schema*. To achieve this goal, a *global schema* of the data is defined, and related to the schemas of the individual sources. Queries are formulated in terms of this global schema. Since the actual data resides in the sources, queries are rewritten into queries over the source schemas, which are then evaluated at the sources. The answers returned from the sources are combined, transformed to be compatible with the global schema, and presented to the user. The integration facilities, namely the global schema, the query translation and query processing algorithms, are performed by a *mediator*, whose main task is to provide users with a unique interface for querying the data. The fact that the sources concern a *restricted domain of interest*, is crucial for the successful deployment of integration systems.

Well-known projects that deal with data integration include Information Manifold [12], Tsimmis [14], Picsel [10], Agora [13] and MIX [3]. As the goal of integration is to support declarative querying and automatic query and result transformations, a number of data integration systems use the well-established tools available for such purposes in the relational model, such as query and transformation languages.

Recently, XML [1] has emerged as the *de-facto* standard for *publishing* and *exchanging* data on the Web. Many data sources export XML data, and publish their contents using DTD's or XML schemas. Thus, independently of whether the data is actually stored in XML native mode or in a relational store, the view presented to the users is XML-based. The use of XML as a data representation and exchange standard raises

⋆ Research supported by grant 018-019 by the Israeli Ministry of Science.

new issues for data integration. A significant issue, as argued in [2], is the inadequacy of XML to serve as a global integration schema.

In this paper we describe an approach to the integration of XML sources, based on the *local-as view* [11] approach to data integration. Our main contributions are as follows: (i) the use of *ontologies* for the global schema; (ii) the definition of a simple but expressive language for *describing* XML resources as *views* of the global schema; (iii) an approach to *query processing*, that includes query *rewriting* from the terms of the global schema into one or more XML queries over the local sources, and (iv) the generation of *query execution plans* that may decompose a single query into queries over multiple sources. The approach has been validated by the ST_YX prototype [9].

The paper is organized as follows : in Section 2 we illustrate the main ideas of the approach by an example. Section 3 presents the *integration data model*, and the *mapping language* for the description of XML resources as views over the global schema. The *query language*, and the *query processing* algorithms are given in Section 4. The ST_YX prototype is sketched in Section 5. Related work is presented in Section 6, and Section 7 presents our conclusions.

2 System Overview

We illustrate our approach via an example dealing with the integration of XML-based information sources on *art and culture*. Formal definitions and technical details are deferred to subsequent sections.

2.1 XML resources

Source S_1, located at *http://www.paintings.com* is an XML resource about painters and their paintings; its XML DTD is illustrated in Fig. 1.

```
<!ELEMENT Painter  (Painting+)>
<!ATTLIST Painter  name CDATA #REQUIRED>
<!ELEMENT Painting EMPTY>
<!ATTLIST Painting title CDATA #IMPLIED
                   year  CDATA #IMPLIED>
```

Fig. 1. XML DTD for source S_1, located at URL *http://www.paintings.com*

The XML DTD for the second source S_2, located at URL *http://www.art.com*, is described in Fig. 2.

As is common in data integration scenarios, a single source may provide only *part* of the information available on a subject. Furthermore, sources differ not only in terms of contents, but also in terms of structure and terminology. Given the hierarchical structure of XML, such differences of structure may be more significant that those that exist in relational sources. For an example of a difference of contents, note that source S_2 might record information on the location of paintings, which is absent in source S_1. As for structure, note that in source S_2 paintings are organized by museums, not by their

```
<!ELEMENT Museum (MuseumName, City, Painting+)>
<!ELEMENT Painting (Title)>
<!ELEMENT MuseumName #PCDATA>
<!ELEMENT City #PCDATA>
<!ELEMENT Title #PCDATA>
```

Fig. 2. XML DTD for source S_2, located at *http://www.art.com*

painters as in source S_1. Consequently, while in the hierarchy of S_1 a painting occurs *below* its painter, in source S_2, if the source were interested in adding the painter for each painting, the painter would occur *below* the painting.

2.2 The Global Schema

The main task of an integration mediator is to provide users with a unique interface for querying the data, independently of its actual organization and location. In our approach, this interface, or global schema, is described as an *ontology*. As used here, an *ontology* denotes a light-weight conceptual model and not a hierarchy of terms or a hierarchy of concepts.

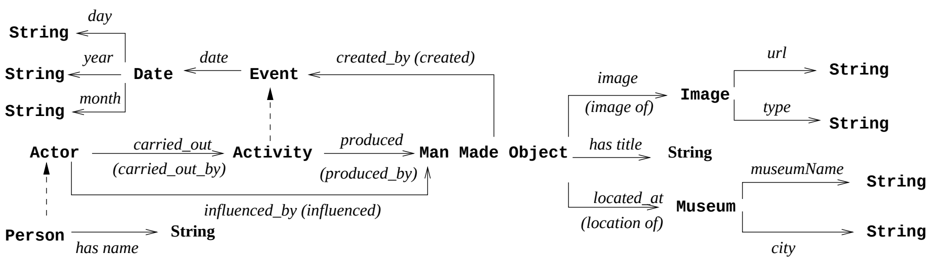

Fig. 3. An Ontology for Cultural Artifacts

Fig. 3 illustrates (part of) a global schema for cultural artifacts inspired by the ICOM/CIDOC Reference Model[3], an international standard for museum documentation. The schema is represented as a *labeled graph.* In this graph, the nodes correspond to *concepts* and *value types*, and the edges depict *roles*, *attributes*, and simple *inheritance* (i.e., isa) links. Roles are binary relations between concepts; attributes connect concepts to value types. Both are depicted by solid arcs. Inheritance (isa) links connect concepts and are depicted by dashed arcs. Each role has an *inverse* depicted in Fig. 3 within parentheses.

The concepts in this schema include Actor, its subconcept Person, and Man_Made_Object. An actor (instance of concept Actor) carries out an activity (instance of concept Activity) to produce a man made object (instance of concept Man_Made_Object).

[3] http://cidoc.ics.forth.gr/crm_intro.html

These relationships are represented by roles *carried_out* and *produced*, respectively. The name of a person (instance of concept Person) is represented by the attribute *has_name*.

The global schema can be viewed as a simple object-oriented data model. Hence, a global schema can be viewed as defining a database of objects, connected by roles, with the concept extents related by subset relationships as per the isa links in the schema. Since it is an integration schema, this is a virtual database. The actual materialization exists in the sources.

Roles can be composed, provided they satisfy certain compatibility constraints. Such compositions are *derived roles*. For example, *carried_out.produced* is a derived role that connects Actor to Man_Made_Object. Combining concepts with (simple or derived) roles induces *derived concepts*. For example, Actor.*carried_out.produced* can be viewed as the sub-concept of Man_Made_Object made of those objects that are reachable from some actor by an instance of this derived role. Both derived roles and derived concepts are referred to as *schema paths*.

The augmentation of the given schema with the derived roles and concepts gives a *derived schema*. It is significant for the integration, since it provides an interpretation for the *mapping rules* (see the following) that describe the sources in terms of schema paths, hence for query processing, as discussed next.

2.3 Mapping Rules

Our integration approach describes XML sources as *local views* on the global schema. Among the different possibilities listed in [7] for defining such mappings, we have chosen the *path-to-path approach*. The description of a source consists of *mapping rules* that associate paths in the source DTD, expressed in *XPath* [6], with paths in the global schema (schema paths). For example, the rules illustrated in Fig. 4 map paths in the source S_1 described in Fig. 1 to paths in the global schema of Fig. 3.

R_1: `http://www.paintings.com/Painter` as u_1	$\rightarrow$ Person
R_2: u_1`/@name` as u_2	$\rightarrow$ has_name
R_3: u_1`/Painting` as u_3	$\rightarrow$ carried_out.produced
R_4: u_3`/@title` as u_4	$\rightarrow$ has_title
R_5: u_3`/@year` as u_5	$\rightarrow$ created_by.date.year

Fig. 4. Set of Mapping Rules for source *http://www.paintings.com*

A rule consists of a name, a left hand side (LHS) and a right hand side (RHS). The LHS contains an *XPath pattern* [6] that starts at a context which is either a concrete URL, as in rule R_1 or a variable, as in rule R_2. The XPath pattern is called the *location path* of the rule. The LHS of a rule also contains a variable declaration (the use of variables will be explained later). The RHS of a mapping rule is a path in the global schema, called the *schema path* of the rule.

Mapping rules define instances of concepts and relationships between them. As an example for the first case, consider the rule R_1 in Fig. 4. It states that the elements of type `Painter`, children of the root elements of the XML documents in S_1 are (descriptions of) instances of concept Person. As an example for the second case, rule R_2 specifies that the value obtained by evaluating XPath pattern `@name` on some XML element x returned by rule R_1, corresponds to a value of attribute *has_name* of x (x is an instance of concept Person). In the same way, rule R_3 connects all instances obtained by rule R_1 to all instances of concept Man_Made_Object obtained by following the path *carried_out.produced*.

This view of mapping rules allows us to define the semantics of XML fragments and their structural relationships in terms of the global derived schema. Thus, R_1 defines a subset of the extent of concept Person, while rule R_3 relates elements in this subset by the derived role *carried_out.produced* to a subset of the extent of Man_Made_Object.

2.4 Query Processing

Users formulate queries on the global schema using a simplified variant of OQL, the standard for querying object databases. For example, here is a query Q_1 that asks for *"titles of the man made objects created by Van Gogh "* :

Q_1: **select** c
from Person a, a.has_name b,
a.carried_out.produced.has_title c
where b = "Van Gogh"

We now discuss the options available for answering such a query, given a set of sources S and mapping rules that relate them to the global schema.

The first, simple, solution is to evaluate this query over each source in S. This means that, given a source $s \in S$, we need to *rewrite* it into an XML query that s can answer. The idea behind this rewriting is the following: Each variable in the query is bound to some schema path. We search for mapping rules or concatenations of mapping rules, that can be used to translate these schema paths to local paths in the source DTD. This is done by *matching* the *schema paths* in the query against the *schema paths* of the mapping rules. A successful matching associates a query variable with a rule, or a concatenation of rules. A *binding* is a vector of such associations for query variables. A *full binding* associates each variable in the query to some rule or concatenation of rules. It can be used to rewrite the query into a query to be evaluated by the XML source.

For example for Q_1 above and for source S_1 we see that instances for variable a are found by rule R_1, for variable b by R_2 and for variable c by the *concatenation* of rule R_3 with rule R_4. The resulting binding is $[a \rightarrow R_1, b \rightarrow R_2, c \rightarrow R_3.R_4]$. By substituting the schema path of each query variable with the location path (LHS) of the corresponding rule, we obtain query $Q_1(a)$:

$Q_1(a)$: **select** c
from `http://www.paintings.com/Painter` a,
a.`/@name` b, a.`/Painting/@title` c,
where b = "Van Gogh"

Query $Q_1(a)$ can be easily translated into the XQuery expression $Q_1(b)$:

```
Q1(b): FOR      $a IN document('http://www.paintings.com')/Painter,
                $b IN $a/@name,
                $c IN $a/Painting/@title
       WHERE    $b = "Van Gogh"
       RETURN   $c
```

Such a matching/rewriting process should be attempted for each source. Then the answers are gathered and returned to the user.

In some cases, however, we cannot obtain a full binding for a given source. Then a second solution for query evaluation is to *decompose* the query into several queries that are evaluated against different sources. Consider the following query Q_2, which asks for *"titles of objects created by Van Gogh, as well as the name and the city of the museum where they are exposed"*:

```
Q2: select d, f, g
    from   Person a, a.has_name b,
           a.carried_out.produced c, c.has_title d,
           c.located_at e, e.museumName f, e.city g
    where  b = "Van Gogh"
```

Source S_1 cannot provide information about the locations of objects, hence there is no mapping rule whose schema path (RHS) matches the schema path *located_at* of query variable e. Thus, we can only obtain a *partial answer* from this source, by evaluating the query $Q_2(a)$ illustrated in Fig. 5. To obtain a full answer we have to join partial answers from different sources.

For the example, the missing information is represented by subquery $Q_2(b)$ illustrated also in Fig. 5, that involves the variables c, e, f, g. The variable c is included in $Q_2(b)$ since it is the *join* variable between the two queries. Thus, we have *decomposed* the initial query into two subqueries $Q_2(a)$ and $Q_2(b)$. Assuming the latter query is successfully evaluated over some source (e.g., source S_2), the results of the two queries are *joined* on c to provide a complete answer to the original query. If such a decomposition cannot be found, the best we can do is to present to the user only the partial results from $Q_2(a)$ evaluated against the first source.

Note that to join two fragments from different sources requires to decide whether the two fragments represent identical objects. *Keys* are introduced to identify objects. In particular, results of queries $Q_2(a)$ over a source S_1 and of $Q_2(b)$ over a source S_2 can be joined only if the same key for man made objects can be provided by these two sources. This implies the use of keys, both in the *global schema* and in the *sources* (the DTD's in Fig. 1 and Fig. 2 do not define such keys). We introduce keys and their usage in Section 3.2.

3 Integration Model

This section is devoted to the detailed presentation of our integration model. Due to space limitations we leave out a detailed discussion concerning the choices of the integration method, and the choice of having a light weight conceptual schema for the mediator schema instead of an XML-based model. A detailed presentation of these

```
Q_2(a): select d                          Q_2(b): select f, g
        from   Person a, a.has_name b             from   Man_Made_Object c,
               a.carried_out.produced c,                 c.located_at e,
               c.has_title d                             e.museumName f, e.city g
        where  b = "Van Gogh"
```

Fig. 5. Queries $Q_2(a)$ and $Q_2(b)$

choices is given in [2]. We first provide a formal definition of the global schema and introduce the notion of derived schema. Mapping rules are described afterwards and we finish this section with a short discussion on keys.

Global Schemas

A global schema is a 6-tuple $\mathcal{S} = (C, R, A, isa, source, target)$, where: (i) C is a set of concepts, (ii) R is a set of typed binary roles connecting concepts in C, (iii) A is a set of attributes of type String[4], (iv) isa is a *binary relationship* between concepts in C, (v) $source$ and $target$ are two typing functions returning for each role/attribute its domain concept and its range concept/type respectively.

A global schema can be represented as a graph of concepts connected by roles. The semantics of a schema is defined by the set of databases that *conform to it*. Each such database contains a set of objects (instances) for each concept in C. These objects are related to each other by instances of roles in R, and to values by instances of attributes in A. Instances of roles and attributes satisfy the typing constraints implied by $source$ and $target$. Roles and attributes are multi-valued and optional. The isa relationship defines a partial order in $\mathcal{S}$, namely a directed acyclic graph. It carries subset semantics and supports role and attribute inheritance. Namely, if $c\ isa\ c'$, then the set of objects of c is a subset of the set of objects of c' and all roles/attributes defined in c' are also defined in c (and its subconcepts). However, if c is not a subconcept of c' such that $source(r) = c'$ for a role r or $source(a) = c'$ for an attribute a, then no object o' of c is related by an instance of r or a to any object, or value respectively. We say that c, c' are *isa-related* if either $c = c'$, $c\ isa\ c'$ or $c'\ isa\ c$.

Finally, we consider schema graphs to be *symmetric* : each role $r \in R$ has an inverse role, denoted r^-, in R. Obviously, $target(r^-) = source(r)$, and $source(r^-) = target(r)$. This is useful for modelling the contents of XML resources as well as for query formulation and, hence, beneficial to have in a conceptual schema.

Schema Paths and Derived Schemas We distinguish two kinds of paths in a schema :

– A *role path* is a sequence of roles $r = r_1 \ \ldots \ r_n$, where for all roles r_i $(1 \leq i \leq n-1)$, $target(r_i)\ isa\ source(r_{i+1})$. Given a role path $r = r_1 \ \ldots \ r_n$, we define its *inverse role path* $r^- = r_n^- \ldots r_1^-$ where r_i^- is the inverse role of r_i

[4] Wlg. we assume that all attributes are of type string; an extension to the types proposed by the XPath model or XML schema should be straightforward.

– A *concept path* p is either of the form c, or a sequence $c.r$, where c is a concept and r is a role path, such that $c\ isa\ source(r)$. The source of p is c and its $target$ is c in the first case, and $target(r)$ in the second case.

The composition of a concept path p and a role path r, denoted $p \circ r$, is well-defined provided that $target(p)\ isa\ source(r)$.

A concept path $p = c.r$ can be viewed as a *derived concept* (denoted by *conc(p)*), standing for *"the instances of $target(p)$ that can be reached from instances of $source(p)$ by following the roles in p, in order"*. Obviously, every concept is also a derived concept.

In the same way, a role path $r = r_1 .\dots. r_n$ can be viewed as a *derived role* (denoted by *role(r)*) connecting instances of concept $source(r_1)$ to instances of concept $target(r_n)$. Similarly to derived roles, we can define *derived attributes*, by a role path followed by an attribute. Like attributes, these do not have inverses. Clearly, every role (attribute) is also a derived role (attribute).

Let $p = c.r$ be a concept path, q be a prefix of r, and $q \setminus r$ denote r with q removed. If c' is either $target(c.q)$ or a superconcept thereof but a subconcept of $source(q \setminus r)$, then $c'.(q \setminus r)$ is called a *suffix* of $c.r$. Obviously, $target(p)$ is a suffix of p.

Given DB, a database of $\mathcal{S}$, we can associate extents with derived concepts in a straightforward manner. We note the following facts concerning these extents. First, the extent of *conc(p)* is a subset of the extent of $target(p)$, hence also of its superconcepts in $\mathcal{S}$. Second, the extent of *conc(p)* is a subset of the extent of each of its suffixes. For example, it is easy to see that p'=Activity.*produced* is a suffix of p=Person.*carried_out-.produced* and all instances of p (objects produced by an activity carried out by a person) are instances of p' (objects produced by activities).

Given a global schema $\mathcal{S}$, the *derived schema* (or *extended schema*) $\mathcal{S}_\mathcal{X} = (C_\mathcal{X}, R_\mathcal{X}, A_\mathcal{X}, source, target, isa_\mathcal{X})$ is defined as follows : (i) $C_\mathcal{X}$ is the set of all derived concepts, defined by the concept paths definable in $\mathcal{S}$; (ii) $R_\mathcal{X}(A_\mathcal{X})$ is the set of all the derived roles (attributes) defined by the role (attribute) paths definable in $\mathcal{S}$, and $source$ and $target$ are defined as above; (iii) the $isa_\mathcal{X}$ relation contains the isa relations from $\mathcal{S}$, and additionally each pair c, c', where c is a derived concept defined by a concept path p in $\mathcal{S}$, and c' is the derived concept defined by a suffix of p.

Our interest in the derived schema is motivated by the fact that some sources may provide data only for derived concepts. The isa relationships in the derived schema enable us to use these sources to provide answers in terms of the original concepts. For example, even if a source provides only information about Person.*carried_out.produced*, this allows us to obtain some instances of Man_Made_Object, although not necessarily all. Note that answers obtained from sources in the *local as view* approach are partial answers in any case.

3.1 Mapping Rules

A source is integrated to the system, by providing a set of *mapping rules* that describe the relationships between the source schema and the global schema. There exist different ways for defining such views varying in terms of size and preciseness of the definition but also in the complexity of the query rewriting algorithm [7]. We have chosen essentially the same approach as in [7], namely to associate paths in the global schema with paths in the source schemas. This allows us to both associate concepts with XML nodes in the sources, and to associate relationships among concepts (expressed as roles or derived roles in the global schema) with XPath *location paths* in the XML sources.

Paths in a source are described in terms of XPath [6] location paths. We assume familiarity with the XPath language. Described in a nutshell, an XPath location path is composed of a sequence of location steps. Location steps have three parts: (i) an *axis* specifies the relationship (child, descendant, ancestor, attribute etc.) between the nodes selected by the location step and the context node, (ii) a *node test* specifies a node's XML type (element, attribute, and so on) and possibly its name, and (iii) *optional predicates* which use XPath expressions to further refine the set of selected nodes.

Let V be a set of variables, and U be a set of URLs. A *mapping rule* is an expression of the form $R : u/q\ as\ v \rightarrow p$, where : (i) R is the rule's *label*; (ii) $u \in V \cup U$, the rule's *root*, is either a variable or a URL (u is called the root of R); (iii) q is an XPath *location path*, called the *location path* of the rule; (iv) $as\ v$ is a *binding* of v (R is called the binding rule of v), where $v \in V$ is a variable; (v) p is a *schema path*. More precisely, it is a role path if u is a variable and a concept path otherwise. A rule R is called a *relative mapping rule* if its root is a variable u, and an *absolute mapping rule* otherwise. In the first case, u is the *root variable* of R, and this occurrence of u is a *use* of the variable. Let $lp(R), sp(R)$ denote R's *location path* and *schema path*, respectively.

Given a set of mapping rules for a source s, we define *reachability* (in s) for rules and variables, as follows : (1) each rule whose root is a URL (the URL of s) is reachable; (2) each variable bound by a reachable rule is reachable; (3) finally, each rule whose root is a reachable variable is reachable. The set of mapping rules is *cyclic* if this definition of reachability leads to a cycle. The simplest case of a cycle is a rule whose left-hand-side contains $v/A\ as\ v$ (provided that v can be reached from a URL by other rules). In this work we consider only acyclic mappings.

A *mapping* M over $\mathcal{S}_{\mathcal{X}}$ and for a source s is a set of mapping rules such that 1) labels are unique (that is, no two rules have the same label), 2) all rules and variables are reachable in M, 3) the concepts, roles and attributes used in its rules occur in $\mathcal{S}_{\mathcal{X}}$ and 4) it contains no cycles.

The concatenation of mapping rules is defined as follows : two rules $R_1 : a/q_1\ as\ v_1 \rightarrow p_1$, $R_2 : v_1/q_2\ as\ v_2 \rightarrow p_2$, can be *concatenated*, if the composition of their schema paths, $p_1 \circ p_2$ is well defined[5]. Note the constraint that the root of R_2 is bound in R_1 and that concatenation is possible only if p_2 is a role path. The result of the concatenation is the rule $R_1.R_2 : a/q_1/q_2\ as\ v_2 \rightarrow p_1 \circ p_2$.

Given a mapping M, its *closure* is the set of all rules that can be obtained from M by repeated concatenation. It is denoted by M^*. Its *expansion*, denoted $\hat{M}$, is the set of absolute rules in M^* ($\hat{M} \subseteq M^*$) and can be computed by a bottom-up fixpoint computation (since we only consider acyclic mappings, we are sure that a finite fixpoint exists).

Given a global schema $\mathcal{S}$, a mapping M over $\mathcal{S}$ can naturally be interpreted in the derived schema $\mathcal{S}_{\mathcal{X}}$. Each absolute rule in $\hat{M}$ defines a derived concept, and each

[5] We do not define any restriction on the concatenation of the rules' location paths.

relative rule in M^* defines a derived role (attribute). Let us denote by $\mathcal{S}_M$ the restriction of $\mathcal{S}_\mathcal{X}$ to the derived concepts, roles and attributes of M^*.

For example, rule $R_1.R_3$ defines a derived concept, *conc(Person.carried_out.-produced)* subconcept of Man_Made_Object, and rule R_3 defines a derived role, *role(carried_out.produced),* between concept Person and concept Man Made Object. Rule $R_3.R_4$ defines a derived attribute *attr(carried_out.produced.has_title)* of concept Person.

A mapping M for a source s associated with URL u, allows us to view a collection of XML fragments reachable from u as a database that conforms to $\mathcal{S}_M$. To define this database, the population of each derived concept, *conc(p)*, is defined as the union of the set of fragments returned by all absolute rules R in $\hat{M}$ where $sp(R) = p$ or p is a suffix of $sp(R)$.

The set of fragments X_R returned by some absolute rule R in $\hat{M}$ is defined as follows. The root of an absolute rule R is the URL u. Hence X_R is assigned the set of XML fragments that can be obtained by applying the location path $lp(R)$ to the XML document identified by u. The set X_R can be computed by a simple fixpoint computation, using rules in M. Since M^* is finite, alternatively the rules of $\hat{M}$ can be used directly.

Similarly, the relative rules of M^* are interpreted as roles (or attributes) of $\mathcal{S}_M$ in this database of XML fragments, represented by location paths.

Before leaving this subject, we note that according to the LAV approach, XML extents defined as above for the concepts are viewed as subsets of the real (but unknown) extents. Indeed, as sources are added, and rules are added to a mapping, the extents grow. In the LAV approach, any set of answers returned for a query is assumed to be a subset of the full (but unknown) answer.

3.2 Keys

As illustrated in Section 2, *keys* are essential to decide whether two XML fragments describe the same concept. We assume that sources are heterogeneous and autonomous, and we do not expect that they provide us with persistent object identifiers that are valid for all sources. The `ID/IDREF` XML attribute mechanisms are used for internal references, but cannot serve for a key mechanism to perform joins between objects that originate from different sources. Sources might specify meaningful keys in terms of XML elements/attributes as proposed in [4, 8, 16], but one cannot expect different autonomous sources to always use the same keys. For example a painting might be identified by its title in one source, by its title and the year of creation in another source.

A way to overcome this problem is to define *global keys* for *concepts* in the global schema. A *key* for a concept c is defined as a list of *derived attributes* (called *key paths*) that originate from concept c and is denoted by $key(c) = \{a_1,\ a_2,\ \ldots\ a_n\ \}$. W.l.g we assume in this paper that a concept is associated with at most one key, and all its subconcepts (including the derived ones) share the same key.

In our global schema, we could state for example, that an instance of concept Person is identified by attribute *has_name*: *key*(Person)=$\{has\_name\}$. Instances of concept Man_Made_Object are identifiable by their title and their year of creation:

key(Man_Made_Object)= $\{has\_title, created\_by.date.year\}$. Images have no key, i.e. key(Image)=$\emptyset$.

4 Query Processing

Our query processing approach is presented in this section. We first introduce the user query language (section 4.1). Two query processing strategies are then discussed. In the first approach (section 4.2), the solution to a query is the union of the complete answers from individual sources. If no complete answer can be obtained from a source, then the source is abandoned. In contrast, the second approach (section 4.3) allows also for incomplete answers from a given source. If a source s can only partially answer a query, then the query is decomposed in two parts one to be fully answered by s and the other part being sent to the other sources. The partial results from different sources are then joined by the mediator using *global keys*.

4.1 Query Language

The users query the virtual database as presented via the global schema, using simple *tree queries*, based on **select-from-where** clauses following an OQL-like syntax. Queries are of the form:

Q: **select** x_i, x_j, ...
from p_1 x_1,
$x_{j_2}.p_2$ x_2, ...
$x_{j_i}.p_i$ x_i, ...
where c_0 **and** c_1 **and** ...

The x_i's are *query variables* and each p_i in the **from** clause is a path in the global schema (schema path), called the *binding path* of x_i and denoted $bp(x_i)$. The first variable x_1 is the *root variable* of the query, and its binding path p_1 is a concept path. For each $i > 1$, there is a single clause $x_{j_i}.p_i$ x_i, and p_i is a role path. We call x_{j_i} the *parent* of x_i. We assume the parenthood relation between variables forms a tree, with x_1 as its root. x_1 ranges over the extent of the derived concept *conc(p_1)*, and $x_i, i > 1$, ranges over the instances defined by traversing instances of the derived role p_i from the instances of its parent.

We assume queries satisfy the following restrictions. First, no restructuring is allowed in the **select** clause. Although this may add expressive power to the language, we feel it is not strictly needed for our application. Certainly, it is orthogonal to the issue of retrieving data from sources, addressed in this paper. Second, the **where** clause is a conjunction of simple predicates, where a simple predicate is of the form $x_i \theta d$ in which $\theta \in \{=, <, >, \leq, \geq\}$ and d is an atomic value. Thus, it is not possible to express joins by equalities between variables, i.e., by predicates of the form $x_i = x_j$. This restricts the expressive power of the query language but simplifies the rewriting and evaluation of queries. Third, schema paths occur in the **from** clause, but not in the **select** clause or the **where** clause of a query. It is easy to show that a query with schema paths in the **select** and the **where** clause can be rewritten into an equivalent query in which they appear only in the **from** clause. Last, the language has no quantifiers, aggregates, or subqueries. However, a variable x_j present in the **from** clause is implicitly existentially quantified. Thus, queries with certain kinds of existential quantification can be expressed in the above form.

Since no joins are allowed in the **where** clause, a query whose variables form a forest can be decomposed into a cross product of several tree queries: the restriction to tree rather than forest queries results in no loss of expressive power.

The result of a such a query is a *set of tuples* of the form $\{[a_i, a_j, \ldots a_k]\}$ where $a_i, a_j, \ldots a_k$ are instances of the variables in the query's **select** clause and can be either atomic values, or XML fragments.

In the sequel, the following representation of tree queries is used. A tree query Q is represented as a labeled tree, $T(Q) = (X, par, bp, ops)$ where X is the set of query variables (tree nodes), par is the parent binary relation between nodes defined above, $bp(x)$ is the binding path of x and ops is a set of operations associated with variable x, defined as follows : for a variable x in the **select** clause $\pi \in ops(x)$, and for each condition $x\theta d$ in the **where** clause, $\sigma_{x\theta d} \in ops(x)$.

4.2 Variable to Rules Bindings

We now proceed to the details of query processing. We first present a simple approach in which a source contributes to the answer only if it can fully answer the query.

To evaluate a query, we need to *rewrite* it into an XML query that some sources can answer. Obviously, in general only some of the sources contain the data requested in the query. Each such source returns a subset of the possible answers; the union of the answers from all relevant sources is presented to the user (see Section 5).

For this rewriting, we use the mapping rules. For a query Q and a source s, we define a *variable to rule binding*, or shortly *variable binding*, as a mapping β from a set of query variables to $M^\star$. We consider only bindings such that $dom(\beta)$ is either empty or is the set of nodes of some prefix[6] of $T(Q)$. The empty binding is denoted by $\beta_\emptyset$. If β binds all variables in Q then it is called a *full binding*, otherwise it is a *partial binding*.

The properties of a binding β are the following : if $dom(\beta)$ is not empty, then β associates each variable in it with a rule of M^*, such that the following hold:

1. if x is the root of query Q, then $\beta(x)$ is an absolute mapping rule such that *conc(sp*$(\beta(x)))$ isa_M *conc(bp*$(x))$, i.e., the derived concept defined by $sp(\beta(x))$ is a subconcept of the derived concept defined by the binding path $bp(x)$ in $\mathcal{S}_M$,
2. else, let $par(x) = x'$, then $\beta(x)$ is a relative rule, and
   - the root variable of rule $\beta(x)$ is bound in rule $\beta(x')$,
   - the role path (RHS) of the rule $\beta(x)$ is equal to the binding path $bp(x)$,
   - and finally, the concatenation of the two rules $\beta(x')$ and $\beta(x)$ is well-defined.

In the first case, x is the root of Q and bound to some (possibly derived) concept by its binding path $bp(x)$ that has the form c or $c.r$. An absolute rule R can provide instances for this concept if its concept path (RHS) $sp(R)$, viewed as a derived concept, is a sub-concept of $bp(x)$ (i.e. if the latter is a suffix of $sp(R)$ or it defines a superconcept

[6] A tree T' is a prefix of a tree T if it is a subtree of T and its root is the same as that of T.

thereof). Note that we use here both derived concepts and the isa relationship between them. Thus, the derived schema defined in Section 3 is essential for our approach to query processing.

In the second case, the assumption that if β is defined on x then it is defined on the parent of x follows from the requirement that its domain is a prefix of $T(Q)$. In this case, the declaration of x in Q has the form $x'.q\ x$, and $bp(x) = q$. Answers for x can be obtained from answers for x', by following the binding path q of x.

A partial binding β is called *maximal* if there does not exist a binding β' such that $dom(\beta) \subset dom(\beta')$ and $\beta(x) = \beta'(x)$ for all x in $dom(\beta)$. It is evident that a full binding is a maximal binding.

Variable Binding Algorithm We will now describe a *variable binding* algorithm $\mathcal{B}(Q, s)$ which takes as an input a query Q and a mapping M for a source s and returns a set of maximal bindings. A binding β is represented as a vector of associations of variables to rules, $[x_1 \mapsto R_1, \ldots x_n \mapsto R_n]$. The algorithm is illustrated in more detail in Fig. 6. First, the variables of the query tree are arranged in pre-order: the root is first, and every other node occurs after its parent. The algorithm starts from the empty binding, and once a set of partial bindings have been constructed, it tries to extend each one, using the ordering of the variables. The extension of a partial binding β by $[x \mapsto R]$ is denoted $\beta \times [x \mapsto R]$.

In the first step, we extend $\beta_\emptyset$ to the root variable x_1. For each absolute rule R in M^* such that the derived concept defined by $sp(R)$ is a subconcept of the derived concept defined by the concept path $bp(x_1)$ in $\mathcal{S}_M$, we create the binding $[x_1 \mapsto R]$ and add it to the set of bindings for x_1. If no absolute rule is found such that the above conditions hold, then the algorithm stops, and returns the empty set. Then, we iterate through the sequence of variables, from the left. Let the current, not yet treated, variable be x_i, and let y be its parent. For each binding β constructed so far, if $y \notin dom(\beta)$ then β cannot be extended to x_i (recall that a binding is always defined on a prefix of $T(Q)$). Else, let binding β associate rule R' with y. Then, for each relative rule R of M^*, such that $bp(x_i) = sp(R)$, if R and R' can be concatenated (i.e, R' binds the variable that is the root of R, and their schema paths can be composed), we extend β by $[x_i \mapsto R]$. In this case, β can be dropped, since the new binding extends it. Note that the edge from y to x_i is *traversed in this step, and only in this step.*

$\mathcal{B}(Q, s)$ finds the *maximal* bindings for a query Q and a mapping M on source s. The proof is straightforward. Consider a binding β in the result set of $\mathcal{B}(Q, s)$. If there exists a variable x that we could add in $dom(\beta)$, this means that there exists some rule R such that $\beta(parent(x)).R$ is well-defined, then by the algorithm x would already be in $dom(\beta)$ which is a contradiction, from the above assumption.

Let us illustrate the algorithm with query Q_2 presented earlier, and the mapping rules for source S_1 illustrated in Fig. 4. Rule R_1 returns answers for variable a. The rule's schema path is Person which is equal to a's binding path (Person). Rule R_2 returns answers for b, since (1) its schema path *has_name* is equal to the variable's binding path, (2) its root variable u_1 is bound in rule R_1, and (3) the composition of the schema paths of rules R_1 and R_2 is well defined, since attribute *has_name* is defined in concept Person. In a similar manner, we find that variable c is bound to rule R_3 and

```
Input :       the sequence of variables of query $Q$, in pre-order: $x_1, \ldots, x_n$;
              the closure of mapping rules $M^*$ of some mapping $M$ for source $s$;
Output :      the set $B$ of maximal bindings for $Q$ and $M$
Algorithm :   $B := \emptyset$;
              for each absolute rule $R \in \hat{M}$
                  if concept path $bp(x_1)$ is equal to or is a suffix of path $sp(R)$
                      /* $conc(sp(R))$ is a subconcept of $conc(bp(x_1))$ */
                      add $[x_1 \mapsto R]$ to $B$;
              for $i = 2, \ldots, n$ {
                  /* $Temp$ contains all maximal bindings up to $x_{i-1}$ */
                  $Temp := B$;
                  $y$ := parent of $x_i$;
                  for each binding $\beta \in Temp$ where $y \in dom(\beta)$ {
                      for each rule $R$ in $M^*$ where $sp(R) = bp(x_i)$
                          if the composition of $\beta(y).R$ is well defined
                              /* $\beta$ is extended to $x_i$ and added to $B$ */
                              add $\beta \times [x_i \mapsto R]$ to $B$;
                      if $\beta$ was extended to $x_i$
                          remove $\beta$ from $B$;
                  }
              }
              return $B$;
```

Fig. 6. Variable binding Algorithm $\mathcal{B}(Q, s)$

variable d to rule R_4. For variable e, we do not find a mapping rule whose schema path is equal to the variable's binding path (*located_at*). The result $\mathcal{B}(Q_2, S_1)$ is the singleton $\{\beta_1\} = \{[a \mapsto R_1, b \mapsto R_2, c \mapsto R_3, d \mapsto R_4]\}$.

4.3 Query Decomposition

Let S be the set of sources mapped to the global schema. Algorithm $\mathcal{B}(Q, s)$ returns for each source s in S the set of maximal bindings. Each such binding β is either full, i.e. $dom(\beta)$ contains all variables in Q (then s can answer the query using β), or *partial*. In the latter case, β provides us with partial answers, i.e. does not provide answers for all variables in the query. To complete these partial answers, we *decompose* the query Q into (i) a *prefix query* that source s can answer using binding β, denoted $Q_p(\beta)$, and (ii) a set of *suffix queries*, denoted $\mathcal{QS}(\beta)$.

As an example, take the result of algorithm $\mathcal{B}(Q_2, S_1)$ calculated for query Q_2, and source S_1 published by the mapping rules illustrated in Fig. 4. It contains a *partial* binding β_1 defined on a proper subset of the variables in the initial query : for an instance of variable c we miss instances for variable e (and its descendants).

To obtain the complete answer, we define (1) a *prefix query* that source S_1 can answer using β (query $Q_p(\beta_1) = Q_2(a)$ illustrated in Fig. 5) and one *suffix* query (query $Q_2(b)$ illustrated in Fig. 5). The prefix query $Q_p(\beta)$ is a prefix of $T(Q)$ and is defined on the set of variables in $dom(\beta)$. The suffix queries of a prefix $Q_p(\beta)$ in Q

are defined as follows. Let N be the set of variables in $Q_p(\beta)$ which contain at least one child in Q but not in $Q_p(\beta)$ (we call N the *boundary* of $Q_p(\beta)$). Then we define a suffix query for each variable x in N as the subtree of Q rooted at x and containing all descendants of x not in $Q_p(\beta)$. It is easy to see that query $Q_2(b)$ illustrated in Fig. 5 is a suffix query of $Q_2(a)$ in Q_2. Observe that for a given prefix query there might exist zero, one or more suffix queries.

Joining the results The results of the prefix query and the suffix queries must be joined. In order to perform the join between a prefix and a suffix query the following two conditions must hold : (1) the concept to which the root of a suffix query is bound, should have a key, and (2) the sources on which the join is performed should provide complete values for this key.

The *key values* of an instance of a concept c are obtained by considering $key(c)$ as a query ranging over all *key paths* in it. The result of a key query is of the form $\{[x_1, x_2, .., x_n]\}$, where the $x_i's$ are instances of the variables to which the key paths are bound in the key query. For example, the query illustrated below returns the key values for an instance x of concept Man_Made_Object :

$key(Man\_Made\_Object)$: **select** t, y
from Man_Made_Object x,
x.has_title t, x.created_by.date.year y

Given a (prefix or suffix) query Q whose result is of the form $\{[a_1, a_2, \dots a_n]\}$ where the $a_i's$ are instances of the variables in the query's **select** clause and a key query $key(c)$, where c is the concept on which the join will be performed, Q is extended to Q' so as to get, as well as the a_i's, the key values for the fragments, instances of concept c, accessed by Q. The result of Q' is of the form $\{[a_1, a_2, \dots a_n, x_1, x_2, \dots, x_p]\}$ where the x_j's are instances of the key query variables (variables bound to the key paths in $key(c)$). For example, the prefix query obtained after extending query $Q_2(a)$ of Fig. 5 by the key of concept Man_Made_Object is given below[7].

$ext(Q_2(a))$: **select** d, t, y
from Person a, a.has_name b,
a.carried_out.produced c, c.has_title d,
c.has_title t, c.created_by.date.year y
where b = "Van Gogh"

Query Execution Plans Let Q be a query and S be a set of sources. A *decomposition* of Q w.r.t. some maximal binding β is a couple $\mathcal{D}(Q, \beta) = [Q_p(\beta), \mathcal{QS}(\beta)]$ such that $Q_p(\beta)$ is a prefix query of Q on source s in S and $\mathcal{QS}(\beta)$ is the set of suffix queries of $Q_p(\beta)$ in Q. For example, $\mathcal{D}(Q, \beta_1) = [Q_2(a), \{Q_2(b)\}]$ is a decomposition for query Q_2 and the maximal binding β_1 defined on source S_1. Observe that $\mathcal{QS}(\beta)$ is empty if β is a full binding for Q.

Let $\mathcal{D}(Q, \beta) = [Q_p(\beta), \mathcal{QS}(\beta)]$ be a decomposition of Q. Then $Q_p(\beta)$ can be translated into a source query using binding β. For each suffix query in $\mathcal{QS}(\beta)$ either

[7] This query can be optimized by keeping the variables that are common to the key query and the prefix query (the case of variables d and t in the example above).

a full binding is found or the suffix query has still to be decomposed. Let Q_i be a suffix query in $\mathcal{QS}(\beta)$ and $\boldsymbol{k_i} = [x_1, \ldots x_p]$ denote the *key* query variables bound to the key paths of concept c_j associated to the root variable of Q_i. Then $Q_p(\beta) \boxtimes_{\boldsymbol{k_i}} Q_i$ denotes the *join* operation between $Q_p(\beta)$ and Q_i (we assume that both queries are extended by the appropriate key queries). A *prefix query rewriting* $\mathcal{LR}(Q, \beta)$ for a decomposition $\mathcal{D}(Q, \beta)$ is defined as the join between a prefix query and all suffix queries Q_i, $1 \leq i \leq n$, in $\mathcal{QS}(\beta)$ (if β is a full binding for Q then $\mathcal{QS}(\beta)$ is empty and $\mathcal{LR}(Q, \beta) = Q_p(\beta)$):

$$\mathcal{LR}(Q, \beta) = Q_p(\beta) \boxtimes_{\boldsymbol{k_1}} \mathcal{GR}(Q_1, S) \boxtimes_{\boldsymbol{k_2}} \ldots \boxtimes_{\boldsymbol{k_n}} \mathcal{GR}(Q_n, S))$$

Then the initial query Q can be rewritten as $\mathcal{GR}$(Q,S) defined as the union of all prefix rewritings for sources in S:

$$\mathcal{GR}(Q, S) = \bigcup_{s \in S} \bigcup_{\beta \in \mathcal{B}(Q,s)} \mathcal{LR}(Q, \beta)$$

Let a *query execution plan* (QEP) be defined as follows: (1) a query q that can be answered by a single source (that is a query for which there exists a full binding) is a(n atomic) QEP; (2) the union of two QEP's is a QEP [8]; (3) the join of two QEP's is a QEP[9]. Basically, sources answer atomic queries in a QEP and the mediator performs joins and unions. A QEP can involve several atomic queries sent to a given source. It might be interesting to combine such queries in a single query. This implies the reorganization using classical properties such as distributivity of union w.r.t. join. Such properties and reorganizations as well as other optimizations are beyond the scope of this paper.

Given a set of sources S and a query Q, the algorithm P(Q) shown in Fig. 7 computes a query execution plan for Q. For each source s and maximal binding $\beta \in \mathcal{B}(Q, s)$, a QEP $P(\beta)$ of the prefix rewriting $\mathcal{LR}(Q, \beta)$ is computed: if β is a full binding (i.e. complete answers are obtained), the result is query Q. Else, if β is a partial binding, then query Q is decomposed into a prefix query $Q_p(\beta)$ and a set of suffix queries $\mathcal{QS}(\beta)$ (these queries are also extended by the key queries as shown before). The query execution plan of Q against source s is obtained by joining $Q_p(\beta)$ with the query execution plan for each suffix query $Q' \in \mathcal{QS}(\beta)$ (variable k' denotes the key query variables of Q'). To calculate the query execution plan of a suffix query Q' the algorithm is called recursively. Finally the obtained plan is added to the existing plan by union.

Observe that there are two reasons to interrupt the calculation of a query execution plan for a given source s and binding β. The most trivial case is that there exists no maximal binding for Q in s. The second reason is that there exists at least one suffix query which cannot be satisfied (empty query execution plan).

[8] Remember that union is heterogeneous, that is two sets of tuples answering the same query but resulting from different sources might have different structures for the i-th component.

[9] We restrict join to the non commutative aforementioned definition of join: the root of the second QEP should belong to the boundary of the first QEP and each of them should correspond to a concept for which a key has been defined.

```
Input:       a query $Q$ and a set of sources $S$
Output:      a query execution plan for $Q$;
Algorithm:   $QEP(Q,S) = \emptyset$;
             for all sources $s \in S$ {
                 if $\mathcal{B}(Q,s) \neq \emptyset$ {
                     /* there exists at least one maximal binding for Q in $s$ */
                     for all bindings $\beta \in \mathcal{B}(Q,s)$ {
                         if $\beta$ is a full binding $P(\beta) := Q$;
                         else { $P(\beta) := Q_p(\beta)$;
                             for all suffix queries $Q' \in \mathcal{QS}(\beta)$
                                 if $P(\beta) \neq \emptyset$
                                     /* there exists a non-empty query plan */
                                     /* for all subqueries up to $Q'$ */
                                     if $QEP(Q',S) \neq \emptyset$
                                         /* there exists a query plan of $Q'$ */
                                         $P(\beta) := P(\beta) \boxtimes_{k'} QEP(Q',S)$;
                                     else $P(\beta) := \emptyset$;
                         }
                         $QEP(Q,S) = QEP(Q,S) \cup P(\beta)$
                     }
                 }
             }
             return $QEP(Q,S)$;
```

Fig. 7. Query Execution Plans Generation

5 System Architecture

In this section we sketch the architecture of the prototype ST_YX [9] (Fig. 8) that implements the data integration approach described previously. XML Web resources can be published on the fly by creating/modifying/deleting mapping rules between source fragments and the global schema using the *Source Publication Interface*. The global schema can be consulted through the *Schema Manager* which is also responsible for its loading in a ST_YX portal. The mapping rules are first validated by the *Rules Manager* which is also responsible for their storage. The publication of a resource also consists in providing an XSLT transformation program [10] that can be used for formatting source data in the query result[11]. *Query processing* is done in several steps: first *user queries* can be formulated using a standard Web browser. They are either created by a generic *Query Interface*, or simply stored in the form of a hypertext link (URL). The *Query Interface* communicates with the *Schema Manager* allowing the user to browse the global schema for the formulation of a query. The *Query Interface* forwards the query to the *Query Parser* which performs a syntactical analysis of the query with some type-checking w.r.t. the global schema and produces a language neutral intermediate repre-

[10] XSL Transformations (XSLT : http://www.w3c.org/TR/xslt)

[11] If the query result contains XML fragments from a source, then those are transformed using the source's XSL Stylesheet.

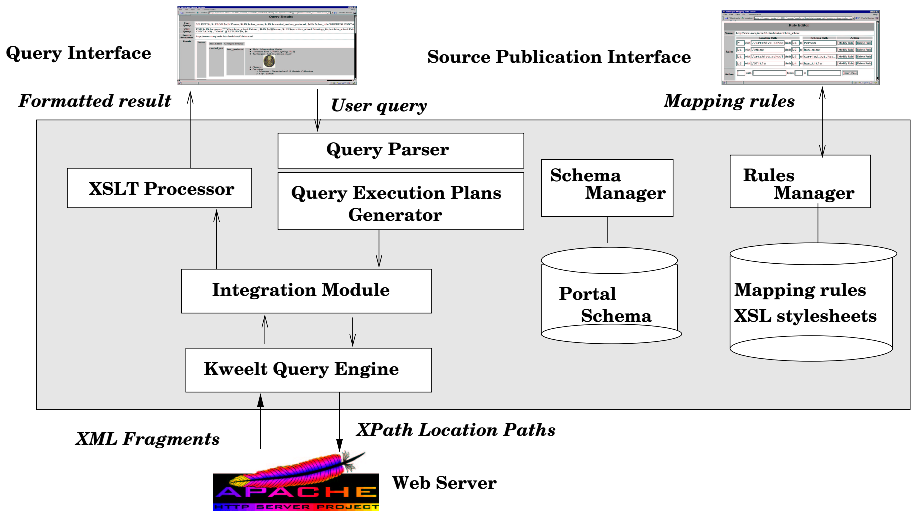

Fig. 8. ST_YX Portal Architecture

sentation of it. The query is then forwarded to the *Query Execution Plans Generator*, which creates the query execution plan. The *Integration Module* rewrites the queries into Quilt Queries and sends them to the *Kweelt Query Engine*[12] for evaluation[13]. The resulting XML fragments are sent to the *Integration Module* that combines the results. This module, based on the query and the mapping rules, inserts schema specific tags, and then the *XSLT Processor* (Cocoon[14]) finally transforms the result into an HTML document which is displayed to the browser of the user.

The ST_YX prototype was implemented in Java JDK 1.2. XML technologies such as XSLT, XPath and the Xalan XML Stylesheet processor[15] were used.

6 Related Work

Data integration has become an important issue during the past years and a large number of integration systems have been proposed. These systems can be classified according to the architectures used for query processing : *data warehouse* systems materialize all source data before query processing, whereas *mediators* propose a virtual database and push queries to the source level based on sophisticated query rewriting algorithms. Our approach clearly belongs to the second category.

[12] Kweelt Query Engine : http://db.cis.upenn.edu/Kweelt/.
[13] The Kweelt query engine can evaluate the subset of XQuery expressions presented in this paper.
[14] http://xml.apache.org/cocoon
[15] http://xml.apache.org/xalan-j/index.html

Mediator systems are classified according to the way sources are described to the mediator and queries are evaluated [11]. Tsimmis [14], MIX [3], YAT [5] and Picsel [10] follow the *global as view* approach and are not directly comparable to ours. On the other hand, Information Manifold [12] follows the *local as view* approach. In this system the global schema is a flat relational schema, and Description Logics is used to represent hierarchies of classes. The sources are expressed as *relational views* over this schema. Query rewriting is done by the *Bucket* algorithm which rewrites a *conjunctive query* expressed in terms of the global schema using the source views. It examines independently each of the query subgoals and tries to find rewritings but loses some by considering the subgoals in isolation. The *MiniCon* algorithm [15] improves the Bucket algorithm by exploiting the input/output dependencies between the query subgoals for reducing the search space of possible rewritings. Algorithm $\mathcal{B}(Q, S)$ presented in this paper resembles to MiniCon since it exploits the *parent/child* dependencies of query variables for query decomposition.

The Agora [13] system, offers an XML view for relational and XML data and user queries are XQuery expressions. Although XML is used as the global data model, an extended use of the relational model is made : the XML view is translated into a generic relational schema, XML resources are described as relational views over this schema and XQuery expressions are translated to standard SQL queries which are then decomposed, optimized and evaluated. Our system and query rewriting algorithm extensively exploit the tree structure of XML data which is described as local views of a more powerful conceptual schema with inheritance.

Last, the Xyleme [7] system is based on a data-warehouse solution for the integration of XML data ("all XML data of the Web"). However, it can be considered as a mediator system, since source data is stored without transformation and users can query this data via different views. Each view is described by a DTD, called *abstract DTD*, and source data is mapped to one or several DTDs using path-to-path mapping rules. These rules are similar to our mapping rules with the difference that they map absolute source paths (starting from the document root) to absolute paths in the abstract DTD (starting from the DTD root element).

7 Conclusions

We proposed in this paper an alternate approach for integrating XML sources following the LAV approach. Instead of choosing for the global view, a relational or XML schema, we advocated the use of an ontology-based mediation. The global schema is close to an object-oriented schema on a terminology describing a common domain of interest and users issue queries on this global schema. Our contributions are (i) a view definition language, (ii) a rewriting algorithm, (iii) an algorithm for generating execution plans, and (iv) a prototype validating the approach.

We are currently working on several extensions concerning our integration model. First, we try to extend the query language by allowing explicit joins in the **where** clause of a query. This does not change the binding algorithm, but increases the complexity of query processing. A second issue we are looking at concerns the usage of *maximal* bindings for query decomposition. In fact, the current version of the rewriting algorithm generates query execution plans which favor information stored locally in the same document. For example, if some source s provides a single full binding for some query Q, the algorithm will return the result of Q in s, but will not try to join s with some other source s'. This restriction can be removed by allowing also partial bindings that are not maximal, but will increase the number of possible decompositions significantly.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data On the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, October 1999.
2. B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Ontology-Based Integration of XML Web Resources. In *International Semantic Web Conference (ISWC)*, Sardinia, Italy, 2002.
3. C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *Demonstrations, ACM/SIGMOD*, pages 597–599, 1999.
4. P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Keys for XML. In *Proc. WWW10*, pages 201–210, 2001.
5. V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of ACM SIGMOD*, Dallas, USA, May 2000.
6. J. Clark and S. DeRose (eds.). XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. http://www.w3c.org/TR/xpath.
7. S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. In *Proc. VLDB*, Rome, Italy, September 2001.
8. W. Fan, G. Kooper, and J. Simeon. A Unified Constraint Model for XML. In *Proc. WWW10*, Hong-Kong, China, May 2001.
9. I. Fundulaki, B. Amann, C. Beeri, and M. Scholl. STYX : Connecting the XML World to the World of Semantics. In *Proceedings of EDBT*, Prague, Czech Republic, March 2002. (Demonstration).
10. F. Goasdoué, V. Lattés, and M-C. Rousset. The use of CARIN language and algorithms for information integration: The PICSEL System. *International Journal on Cooperative Information Systems*, 2000.
11. A. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4):40–47, 2000.
12. A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proc. VLDB*, pages 251–262, Mumbai (Bombay), India, September 1996.
13. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Hterogeneous Data Sources. In *Proc. VLDB*, Rome, Italy, September 2001.
14. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. ICDE*, pages 251–260, Taipei, Taiwan, March 1995.
15. R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proc. VLDB*, pages 484–495, Cairo, Egypt, September 2000.
16. H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Recommendation, May 2001. http://www.w3.org/TR/XML-schema-1.