Answering XML Queries over Heterogeneous Data Sources

Ioana Manolescu
INRIA, Rocquencourt
Ioana.Manolescu@inria.fr

Daniela Florescu
Propel
Daniela.Florescu@propel.com

Donald Kossmann
Technical University of Munich
kossmann@informatik.tu-muenchen.de

Abstract

This work describes an architecture for integrating heterogeneous data sources under an XML global schema, following the *local-as-view* approach (local sources' schemas are described as views over the global schema). In this context, we focus on the problem of translating the user's query against the XML global schema into a SQL query over the local data sources.

1 Introduction

In recent years, there have been many research projects focusing on logical data integration; among them we cite Garlic [17], the Information Manifold [11], Disco [21], Tsimmis [8], and Yat [1]. The goal of such systems is to permit the exploitation of several independent data sources as if they were a single source, with a single global schema. A user query is formulated in terms of the global schema; to execute the query, the system translates it into subqueries expressed in terms of the local schemas, sends the subqueries to the local data sources, retrieves the results, and combines them into the final result provided to the user. Data integration systems can be classified according to the way the schema of the local data sources are related to the global, unified schema. A first approach is to define the global schema as a view over the local schemas: such an approach is called *global-as-view* (GAV). The opposite approach, known as *local-as-view* (LAV) consists of defining the local sources as views over the global schema.

The tradeoffs between LAV and GAV (as presented in [10]) are the following. In the GAV approach, translating the query on the global schema into queries on the local schemas is a simple process of view unfolding.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

In the case of LAV, the query on the global schema needs to be reformulated in the terms of the local data sources' schemas; this process is traditionally known as "rewriting queries using views" and is a known hard problem [13, 10]. On the other hand, in a GAV architecture, to handle modifications in the local data sources set or in their schemas, the new global schema needs to be re-designed considering the whole modified set of sources. In a LAV architecture, a local change to a data source can be handled locally, by adding, removing or updating only the view definitions concerning this source; therefore, LAV scales much better. Also, if the local data sources do not have the same data format (e.g., some are relational while others are XML), it would be difficult to define the global schema as a view over sources in different formats; in contrast, using LAV, each source can be described in isolation, by a view definition mechanism appropriate to its format.

Nowadays, the popularity of XML as a data exchange format makes it a good candidate for the global schema in data integration applications. Furthermore, using an XML-based schema at the interface level allows to hide the proprietary schemas that the data owners do not want to disclose, and to adhere to a newly-established standardized interface without having to migrate existing data. While XML is an interesting option for a global schema format, for many application domains, standardized, domain-specific XML global schemas have already been established. These standardized schemas, available as DTDs or XML Schemas, provide the basis for large-scale integration applications, for which LAV is preferable.

In this work, we present a methodology for integrating data sources of diverse formats, including XML and relational, under an XML global schema, using the LAV approach. Our approach is implemented in the Agora data integration system [15]. In Agora, relational and tree-structured data sources are defined as views over the global XML schema, by means of an intermediate *virtual, generic, relational schema*, closely modeling the generic structure of an XML document.

This paper is organized as follows. In section 2, we detail the context of our work, outline our architecture, and briefly present XQuery, the XML query language

```
Med.XML
<medical>
  <patient ssNo="123">
    <name>"Doe,John"</name> <dob>"1/1/1960"</dob>
    <address>"1, South St., Palm Beach, FL"</address>
  </patient>
  <patient SSno="101"><name>"Ale, Mary"</name>
    <dob>"2/6/1970"</dob>
    <address>"2,Pine Rd., Bear Canyon, MN"</address>
  </patient>
 <record><patientSSno>"123"</patientSSno>
  <entry entID="1"><date>"1/9/90"</date>
    <symptoms>"fatigue, bad sleep"</symptoms>
    <diagnosis></diagnosis>
    <medication>"blood tests"</medication>
  </entry>
  <entry entID="2" rel_previous="1"><date>"10/9/90"</date>
    <symptoms>"low blood iron"</symptoms>
    <diagnosis>"Anemy"</diagnosis>
    <medication>"Biofer once a day"</medication>
  </entry>
 </record>
</medical>
```

```
for $no in distinct(document("med.xml")//record/@ssNo)
let $recs:=document("med.xml")//record[patientSSNo=$no]
return <pollutionIncident>$no,
          (for $e in $recs
           where $e/date > "1/1/91" and
                 contains($e/diagnosis, "pollution")
           return $e/diagnosis)
       </pollutionIncident>
```

Figure 1: Sample XML document with medical data (top) and user query (bottom).

used in Agora [24]. We then describe the query processing steps that are applied to an XQuery query. Section 3 provides normalization rules that make the query easier to translate on the generic schema, or signal the fact that the translation is unfeasible, due to the expressive power mismatch between XQuery and SQL. Section 4 shows how to translate normalized XQuery queries into SQL queries on the generic schema, and section 5 discusses the rewriting of the SQL query on the generic schema into a SQL query on the real data sources. Section 6 explains how we can enlarge the translatable subset of XQuery by allowing intermediate XML query results; related work is discussed in section 7, and we conclude in section 8.

The XQuery language is still work in progress, and our query translation methodology is valid with respect to the syntax and semantics defined as of February 2001. Advances in the standardization process may slightly change the semantics of the language; our query translation method from XQuery to SQL is to be considered modulo these possible changes.

2 XML data integration methodology

2.1 Problem definition

Our goal is to integrate relational data and DOM-compliant data sources under a global XML schema. DOM (Document Object Model) is a generic API that allows the manipulation of tree-structured documents, in particular HTML and XML [23]. We designate by "DOM data source" any source supporting the DOM interface, regardless of its storage mechanism. Our data integration methodology must allow for efficient query processing, in particular by exploiting as much as possible the query processing capabilities of the local data sources, be they relational or DOM-compliant.

The query language that our mediator supports is XQuery, the standard XML query language being elaborated by the W3C [24]. The XQuery data model views an XML document as a labeled tree with references; its type system follows that of XSchema. Besides value and node types, the data model considers only ordered lists; a significant general feature of the algebra is the automatic *list flattening* - lists of lists are always unnested [22]. XQuery has static and dynamic semantics, according to the way typechecking is performed; for the purpose of this paper, we always consider dynamic semantics.

XQuery is centered on the notion of *expression*; starting from constants and variables, expressions can be nested and combined, using arithmetic, logical and list operators, navigation primitives, function calls, higher order operators like *sort*, conditional expressions, element constructors etc. For navigating in a document, XQuery uses path expressions, whose syntax is borrowed from the abbreviated syntax of XPath. The evaluation of a path expression on an XML document returns a list of information items, whose order is dictated by the order of elements within the document (also called *document order*). XQuery provides a *range* predicate whose meaning is also based on order: *E[range n to p]* evaluates the expression E, yielding a list, and selects from this list the sublist of the n-th to p-th items. The precise semantics of path expressions is still under discussion; in this paper, we consider a snapshot of the semantics for simple path expressions, as it was in February 2001. Since the semantics of arithmetic and boolean operators is also being currently discussed, in this paper we interpret them following simple SQL semantics.

A powerful feature of XQuery is the presence of FLWR expressions (for-let-where-return). The *for-let* clause makes variables iterate over the result of an expression or binds variables to arbitrary expressions, the *where* clause allows specifying restrictions on the variables, and the *return* clause can construct new XML elements as output of the query. In general, an XQuery query consists of an optional list of namespaces definitions, followed by a list of function definitions, followed by a single expression.

2.2 Motivating example

Our sample data sources are inspired from the domain of health care. Figure 1 shows the data presented to the user under the form of a single XML document, containing both administrative information about patients in the *patient* elements, and medical files that physicians keep on patients, represented by *record* elements. The global schema consists of this document's DTD. Data is actually stored in two local sources: administrative information is stored in a relational format as a table Patient(name:string, dob:Date, SSno: integer, address: string), while *record* elements with medical data are stored as such in a separate XML file.

The user query that we consider is shown at the bottom of figure 1. In this query, *$no* iterates over all *ssNo* attributes of *record* elements, and *$recs* is successively bound to the list of records whose *ssNo* attribute value is equal to that of *$no*. For each value of *$no*, a new *pollutionIncident* element is created, containing only those records in *$recs* which are less than 10 years old and whose diagnosis contains the word "pollution".

2.3 Data integration methodology

In the Agora integration system, we adopt the following solution to the issues presented in section 2.1. For efficiency, query optimization and most of query execution are carried on according to the relational model and algebra. Agora is built on top of the LeSelect relational data integration engine [12]. LeSelect has a distributed peer-to-peer architecture; relational data sources are published on a LeSelect mediator by registering them with a data wrapper connected to the mediator. An user query is formulated in SQL, and it is optimized and executed in a distributed manner, involving the wrappers of all data sources in the query, and possibly their corresponding mediators.

To enable LeSelect's execution engine and optimizer to process DOM data sources, Agora provides a way of exploiting such sources as a collection of tables. The DOM interface provides a set of API calls for accessing the content of a document; a special wrapper designed for DOM-compliant data sources exports to the mediator one virtual table for each such API call. A complete scan of a virtual table exported by the DOM wrapper is generally not possible, since some input parameters are required for each DOM call. In our system, such restrictions are modeled by *binding patterns*, and the DOM wrapper is capable of processing SQL subqueries with binding patterns on the virtual tables that it exports. To handle restricted access tables, LeSelect's optimization algorithm follows a variant of dynamic programming enhanced with binding patterns [7].

To execute XQuery queries via LeSelect's relational engine, we devised a query translation methodology that proceeds in three steps, shown in figure 2. First, the query is **normalized**, applying equivalent transformations that bring it to a syntactical form which can be directly translated to SQL, if this is possible. The normalized query is **translated** into a SQL query on a generic, virtual, relational schema. This schema, detailed in section 4, is used only as an intermediate layer; it is never materialized as such, and is invisible to the system's users. This first translation step is completely independent of the relation between the virtual XML global schema and the real data sources; it only gets the query across the language gap. Finally, the SQL query on the generic schema is **rewritten** into a SQL query on the real data sources. In this relational query rewriting step, we use the definitions of the data sources as views over the virtual generic schema.

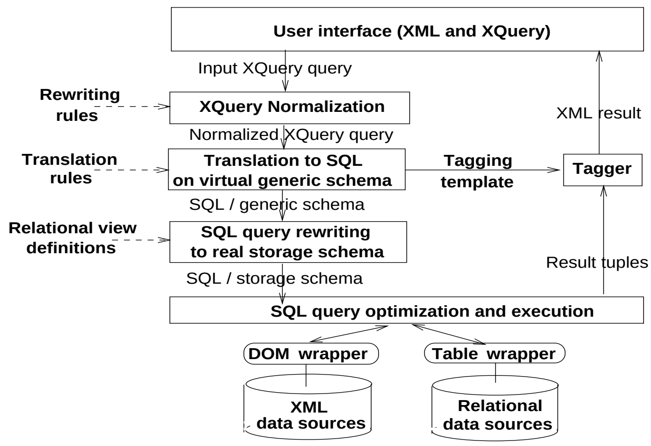

Figure 2: General architecture of the Agora data integration system.

Not all features of XQuery can be translated to SQL; there are two distinct sources of difficulties. First, some of the language's features do not have SQL equivalents due to a semantic mismatch between the two models; such features are identified (and the translation fails) during the normalization phase. Second, for those XQuery queries that could be brought to a SQL form on the virtual generic schema, relational query rewriting might fail, because state-of-the-art query rewriting algorithms for SQL semantics do not handle well arbitrary levels of nesting, grouping etc. We stress the fact that these difficulties are not due to our translation methodology; we merely separated the language-dependent translation step, transforming an XQuery query into SQL on the generic schema, from the rewriting step reformulating the query in terms of relevant data sources. This separation allows us to provide independent solutions for the two steps, and to distinguish among the two sources of difficulties.

If the rewriting step succeeds, we obtain a SQL query referring to well-identified local data sources, either relational or DOM-compliant. Tuples resulting from the relational execution of this query are treated by a tagger module, that structures them into the desired XML format of the result. This structure information is produced during the translation step and is passed directly from the translator to the tagger under the form of a tagging template, as shown in figure 2. The tagger's functioning is inspired by work done in [20].

3 XQuery normalization

In this section we use the following notations. Lower case letter like x, y, z correspond to individual XQuery query variables, while capital letters like E, R, C denote XQuery expressions. We denote simple path expressions by PE, and element constructor expressions by EC. For brevity, we some times write a single for clause "for $\overrightarrow{x}$ in E" instead of "for x_1 in E_1, x_2 in $E_2(x_1)$, … x_n in $E_n(x_1, \dots x_{n-1})$"; in this case, E is an expression of arity n, and $\overrightarrow{x}$ are consecutively bound to each tuple of values that result from E's evaluation. Using these notations, the classes of translatable queries can be informally described as follows:

- simple path expressions, starting with a document node or with an implicit context node, consisting of steps of the following kinds: child, descendent, attribute, and dereferencing, and eventually interspersed predicates.
- element constructors whose tags and data are either constants or come from simple path expressions as described above, or from translatable FLWR expressions;
- translatable FLWR expressions of the form *for* $\overrightarrow{x}$ *in* E *where* $C(\overrightarrow{x})$ *return* $R(\overrightarrow{x})$, where: E denotes a n-uple of simple path expressions, $C(\overrightarrow{x})$ is a logical expression constructed with simple path expressions depending on $\overrightarrow{x}$ and usual operators; $R(\overrightarrow{x})$ is a list of simple path expressions depending on $\overrightarrow{x}$, or a translatable element constructor;
- arithmetical and logical expressions on scalar types.

3.1 Normalization rules

In this section, we provide several equivalence rules to simplify the user's query and bring it to one of the translatable forms, when possible.

Let clauses are treated as temporary variable definitions. During normalization, they are eliminated as shown in rule NR_1: the expression binding the variable y is substituted to all its occurrences. Non-recursive function definitions are eliminated; calls to such functions are replaced with the body of the function, applying the proper substitutions.

NR$_1$	for $\overrightarrow{x}$ in E_1 let $y = E_2(\overrightarrow{x})$ for $\overrightarrow{z}$ in $E_3(\overrightarrow{x}, y)$ where $C(\overrightarrow{x}, y, \overrightarrow{z})$ return $R(\overrightarrow{x}, y, \overrightarrow{z})$	$\Rightarrow$	for $\overrightarrow{x}$ in E_1, $\overrightarrow{z}$ in $E_3(\overrightarrow{x}, E_2(\overrightarrow{x}))$ where $C(\overrightarrow{x}, E_2(\overrightarrow{x}), \overrightarrow{z})$ return $R(\overrightarrow{x}, E_2(\overrightarrow{x}), \overrightarrow{z})$
	function $f(\overrightarrow{x})${ return $E(\overrightarrow{x})$ } Q	$\Rightarrow$	$Q[f \leftarrow E]$

In XQuery, FLWR expressions can be used as building blocks for more complex expressions. Rule NR_2 unnests expressions of the form $E_1(FLWR)$, in the case when expression E_1 distributes over list concatenation, e.g. E_1 is a child path step (illustrated under the rule). This rule is a consequence of the automatic list flattening feature of the XQuery algebra. Rule NR_2 does not hold if E_1 is, for example, a *range* operator, or an aggregate function.

```
for r in document("med.xml")//record,
    x in r/entry
where x/date="1/9/90"
return
    (for y in documents("med.xml")//patient
     where r/patientSSno=y/@SSno return y)
                     ⇒
for r in document("med.xml")//record,
    x in r/entry,
    z in (for y in documents("med.xml")//patient
          where r/patientSSno=y/@SSno
          return y)
where x/date="1/9/90" return z
```

Figure 3: Example of unnesting return clauses.

NR$_2$	E_1(for $\overrightarrow{x}$ in E_2, where $C(\overrightarrow{x})$ return $E_3(\overrightarrow{x})$)	$\Rightarrow$	for $\overrightarrow{x}$ in E_2, where $C(\overrightarrow{x})$ return $E_1(E_3(\overrightarrow{x}))$
	(for $\overrightarrow{x}$ in E_2 where $C(\overrightarrow{x})$ return $E_1(E_3(\overrightarrow{x}))/nameTest$	$\Rightarrow$	for $\overrightarrow{x}$ in E_2 where $C(\overrightarrow{x})$ return $E_3(\overrightarrow{x})/nameTest$

Element constructors nested within path expressions have the general form $PE(EC(\overrightarrow{x}))$, where $\overrightarrow{x}$ represent variables that may have been bound outside this expression. If PE consists of path steps without the *range* predicate, the path steps can be composed with the element constructor and the expression rewritten, so that the element constructor disappears. Rule NR_3 shows how to push such steps into element constructors, when $E(\overrightarrow{x})$ evaluates to a list of XML elements; the comma represents list concatenation. If the element constructed by the expression $EC(\overrightarrow{x})$ has text children, they are erased by the translation. A simple similar rule holds for attribute steps.

NR$_3$	(<tag> $E(\overrightarrow{x})$ </tag>)//nameTest	$\Rightarrow$	if tag = nameTest then (<tag>$E(\overrightarrow{x})$</tag>, $E(\overrightarrow{x})$//nameTest) else $E(\overrightarrow{x})$//nameTest
	(<tag> $E(\overrightarrow{x})$ </tag>)/nameTest	$\Rightarrow$	for y in $E(\overrightarrow{x})$ where name(y)= nameTest return y

Rule NR_4 unnests FLWR expressions nested within the for clause of an outer FLWR expressions.

NR$_4$	for $\overrightarrow{x}$ in E_1, y in (for $\overrightarrow{z}$ in $E_2(\overrightarrow{x})$ where $C_1(\overrightarrow{x}, \overrightarrow{z})$ return $E_3(\overrightarrow{x}, \overrightarrow{z})$) where $C_2(\overrightarrow{x}, y)$ return $E_4(\overrightarrow{x}, y)$ $\Rightarrow$ for $\overrightarrow{x}$ in E_1, $\overrightarrow{z}$ in $E_2(\overrightarrow{x})$ where $C_1(\overrightarrow{x}, \overrightarrow{z})$ and $C_2(\overrightarrow{x}, E_3(\overrightarrow{x}, \overrightarrow{z}))$ return $E_4(\overrightarrow{x}, E_3(\overrightarrow{x}, \overrightarrow{z}))$

Rule NR_5 unnests FLWR expressions nested in the return clause of another FLWR expression. This rule is valid because of the implicit list flattening of the algebra; such a rule would not hold in OQL.

NR$_5$	for $\overrightarrow{x}$ in E_1 where $C_1(\overrightarrow{x})$ return (for $\overrightarrow{y}$ in $E_2(\overrightarrow{x})$ where $C_2(\overrightarrow{x}, \overrightarrow{y})$ return $E_3(\overrightarrow{x}, \overrightarrow{y})$)	$\Rightarrow$	for $\overrightarrow{x}$ in E_1, z in (for $\overrightarrow{y}$ in $E_2(\overrightarrow{x})$ where $C_2(\overrightarrow{x}, \overrightarrow{y})$ return $E_3(\overrightarrow{x}, \overrightarrow{y})$) where $C_1(\overrightarrow{x})$ return z

Rule NR_6 unnests complex expressions built on top of conditional expressions. NR_6(a) is meant for cases when E is constructed only with simple path expression steps, element constructors, or arbitrary function calls.

NR$_6$ (a)	$E(\text{if } C(x) \text{ then } E_1(x) \text{ else } E_2(x))$ $\Rightarrow$ $\text{if } C(x) \text{ then } E(E_1(x)) \text{ else } E(E_2(x))$
NR$_6$ (b)	for $\overrightarrow{x}$ in E_1, y in (if $C_1(\overrightarrow{x})$ then $E_2(\overrightarrow{x})$ else $E_3(\overrightarrow{x})$) where $C_2(\overrightarrow{x}, y)$ return $E_4(\overrightarrow{x}, y)$ $\Rightarrow$ for $\overrightarrow{x}$ in E_1, y in $E_2(\overrightarrow{x})$ where $C_1(\overrightarrow{x})$ and $C_2(\overrightarrow{x}, y)$ return $E_4(\overrightarrow{x}, y)$ $\cup$ for $\overrightarrow{x}$ in E_1, y in $E_3(\overrightarrow{x})$ where $\neg C_1(\overrightarrow{x})$ and $C_2(\overrightarrow{x}, y)$ return $E_4(\overrightarrow{x}, y)$

NR$_6$(b) shows how to eliminate conditional expressions directly nested within a *for* clause; note that this rule modifies the order of the result, therefore it can be applied only if the order of the result is not important. For brevity, we omit the rules unnesting conditional expressions within *where* or *return* clauses, and conditional expressions; we refer the reader to [14].

Rule NR$_7$ performs a simple syntactic transformation: if E_2 is a predicate restricting the result of E_1's evaluation, the path predicate notation can be replaced with a where clause in a FLWR expression, since the test has existential semantics in both cases. As an application, path predicates in the for clause of a FLWR expression can be moved to the where clause; we denoted by $/PE$ the final part of the simple path expression x iterates over.

NR$_7$			
	$E_1[E_2]$	$\Rightarrow$	for x in E_1 where $E_2(x)$ return x
	for x in $E_1[E_2]/PE$ where $C(x)$ return $R(x)$	$\Rightarrow$	for y in E_1, x in y/PE where $C(x)$ and $E_2(y)$ return $R(x)$

Untranslatable features of XQuery

Various features of XQuery are difficult or impossible to translate to SQL, no matter what relational schema is used for the target query, because the inner logic of these language features is incompatible with the semantics of SQL. Examples of XQuery expressions that pose difficulties are: scalar constant expressions, run-time access to an element's type (*instanceOf*, *type switch*, *cast*, *treat*), non-linear recursion, heterogeneous type unions, and identity-based operations.

Document order-preserving operators and the *range* predicate deserve a special discussion. A first thing to note is that the order of the result in a simple XQuery expression, without nesting, may come only from some document or data order, perhaps from a cross-product of such orders. It is possible to capture the result order of such a simple expression by an SQL query, but this query involves aggregation and its rewriting is not trivial [14]. Thus, even if the document order is within the expressive power of SQL, operators related to order make query translation cumbersome or may even make it fail. Second, note that in XQuery, order can appear at any level of nesting within a complex expression, while in SQL this is only possible at top-level: therefore, correctly translating a nested order-conscious XQuery query by a single SQL query is impossible. To execute such queries by a relational framework, one needs to make several passes, materializing intermediate XML

Document(**docID**, docURIID, rootElemID) URI(**uriID**, uriValID) ProcInstr(**piID**, piVal1ID, piVal2ID) QName(**qNameID**, qnPrefixID, qnLocalID) Attribute(**attrID**, attrElID, attrNameID, attrValID) Element(**elID**, elQNameID, elTypeID) NameSpace(**nsID**, nsValID, nsURIID) Comment(**commID**, commValID) Value(**valID**, value) Child(**parentID**, childID, childValID, **childIndex**)
TransClosure(parentID,childID)

Figure 4: Virtual generic relational schema representing information from an XML document.

results and running a sequence of XQuery queries, as we show in section 6.

4 Translating normalized XQuery into SQL

Queries within the normalized subset of XQuery are transformed in SQL queries on the real data sources in two steps: first, they are translated into SQL queries on the virtual generic schema in figure 4, then, by a relational query rewriting step, they become SQL queries on the local data sources schema. In this section, we detail the translation step, which does not yet take into consideration the schemas of the local sources.

4.1 Virtual generic schema as support for translation

The simple generic, virtual, relational schema that we use is shown in figure 4; in each table, primary keys are in bold characters. This schema is constructed as a fully normalized relational version of the hierarchical structure of an XML document; foreign keys represent the relationships between different entities within a document. The last table, TransClosure, is redundant; it represents the transitive closure of the parent-child relationship modeled by the Child table. This table is useful for translating recursive XML path expressions, as described in section 4.2, and for rewriting the resulting queries, as shown in section 5.3; we stress the fact that it is *virtual*, i.e. it does not need to be materialized or maintained.

Using the virtual generic schema has several advantages. First, it connects the relational (and other) data sources and the XML global schema. This schema represents a middle ground for query translation: it is a minimal lossless schema with respect to the information contained in an XML document. Since this generic schema does not lose any of an XML document's information content, XQuery constructs that cannot be translated to it cannot be translated to any relational schema, simply because their semantics cannot be adapted to the semantics of SQL. At the same time, it is a middle ground for view definitions: data sources described as views over this generic relational schema are in fact defined in terms of the global XML schema, thus following the LAV technique.

To handle the translation of XQuery constructs referring to a document order, we assume that among elements belonging to the same document, the elID virtual field in the virtual schema reflects this order. To actually return query results in correct document order, all data sources must provide the correspondent of an order-reflecting element ID.

4.2 Translating simple path expressions

Let us denote by $T(E) = (S(E), F(E), W(E))$ the translation function that, for a given expression E, computes the select, from and where parts of the corresponding SQL query.

Rule TR_1 translates simple path expressions denoting a document root. E may be either a string constant, or a more complex XQuery expression, whose SQL translation is a row subquery returning one string:

$\textbf{TR}_1$	T(document(E))= select d.docID from Document d, URI u, Value v where d.docURIID=u.uriID and u.uriValId=v.valID and v.value=$T(E)$

The following rules show how to translate path expressions, given the translation of the path shorter by one step. TR_2 shows how to add a final "child" step to the SQL translation of an expression; again, there are two slightly different cases, according to the name test being a constant or resulting from a complex expression. We show the rule for the most general case; if E_2 is a constant, simply replace $T(E_2)$ with the constant. Since the path expression is correctly typed, we know that $S(E_1)$ must be an element ID, and that $T(E_2)$ must return a single row with one string column.

$\textbf{TR}_2$	$T(E_1/E_2)$ = select e.elID from $F(E_1)$, Child c, Element e, Qname q, Value v where $W(E_1)$ and c.parentID=$S(E_1)$ and c.childID=e.elID and e.elQNameID=q.qNameID and q.qnValID=v.valID and v.value=$T(E_2)$

We move on to translate the expressions whose final step is a "descendent" step, denoted by "//". Note the use of the TransClosure table to express arbitrary depth nesting.

$\textbf{TR}_3$	$T(E_1//E_2)$ = select e.elID from $F(E_1)$, TransClosure tc, Element e, QName q, Value v where $W(E_1)$ and $S(E_1)$=tc.parentID and tc.childID=e.elID and e.elQNameID=q.qNameID and q.qnLocalID=v.valID and v.value=$T(E_2)$

Rule TR_4 shows how to translate a final "attribute" step; this rule also has two variants, depending on whether the attribute name is a string constant or results from a different expression.

$\textbf{TR}_4$	$T(E_1/@attName)$ = select a.attrID from $F(E_1)$, Attribute a, Value v where $W(E_1)$ and a.attrElID=$S(E_1)$ and a.attrNameID=v.valID and v.value=$T(attName)$

Rule TR_5 translates a dereferencing step. Note that in the SQL translation, the query translator has inserted the name of the ID attribute in the target element, *id*, although it was not supplied in the original XQuery expression; this information is taken from the DTD of the document being queried. We only show the case when the attribute name is a constant; if it results from a more complex expression, the corresponding subquery would replace *attName* in the translation.

$\textbf{TR}_5$	$T(E_1/@attName \rightarrow elName)$ = select e.elID from $F(E_1)$, Attribute a1, Value v1, Value v2, Element e, QName q, Value v3, Attribute a2 Value v4, Value v5 where $W(E_1)$ and a1.attrElID=$S(E_1)$ and a1.attrNameID=v1.valID and v1.value=$attName$ and a1.attrValID=v2.valID and e.elQNameID=q.qNameID and q.qnLocalID=v3.valID and v3.value=$elName$ and a2.attrElID=e.elID and a2.attrNameID=v4.valID and v4.value=id and a2.attrValID=v5.valID and v2.value=v5.value

In general, the results of path expressions should come in document order; SQL queries, however, do not guarantee result order, unless an explicit ORDER BY clause is added. Since we require element IDs to reflect document order, to correctly order the translation results, one only needs to add, for example, to $T(E_1/E_2)$, "order by $S(E_1)$, e.elID". Even if $T(E_1)$ was already sorted on $S(E_1)$, after the extra joins the ordering needs to be re-established.

4.3 Translating FLWR expressions

Recall that in a FLWR expression, the *for* clause produces tuples of bindings for the variables in the query, the *where* clause poses conditions that discard some of these tuples, and the *return* clause uses the tuples of bindings that satisfy the selection conditions to construct the result, either under the form of complex structured XML elements or as tuples of flat values.

Rule TR_6 translates a simple FLWR expression, whose *for* and *where* clauses contain only simple path expressions, and that returns all the variables bound in *for-where*. Figure 5 shows a translation example.

$\textbf{TR}_6$	T(for x_1 in E_1, x_2 in $\text{E}_2(x_1)$, ... x_n in $E_n(x_1, \ldots x_{n-1})$ where $C(x_1, \ldots x_n)$ return $x_1, \ldots x_n$) = select $S(E_1), S(E_2) \ldots S(E_n)$ from $F(E_1), \ldots F(E_n)$ where $W(E_1)$ and ... and $W(E_n)$ and exists $T(C(x_1, \ldots x_n))$

To respect the semantics of XQuery, the evaluation of such a path expression should result into $(x_1, x_2, \ldots, x_n)$ tuples sorted in the lexicographic order derived from the order in each E_i. From a database point of view, ignoring the order would result in more efficient execution plans. If the order of tuples is important, a final *sort by* x_1 *asc, ...,* x_n *asc* is added.

To explain the translation of queries returning newly constructed XML elements, we first show how to translate a single element constructor. An element constructor appearing in an XQuery query may depend on variables that have been previously bound in the query. To correctly structure and order the information needed in order to build an XML element, we borrow the *sorted outer union* approach presented in [20]. Translation

T(for x_1$ in document("med.xml")//entry, x_2$ in x_1$/date where x_2$="1/9/90" return x_1$, x_2$) =	select e1.elID, e2.elID from Document d, URI u, Value v1, TransClosure tc, Element e1, QName q1, Value v2, Child c1, Element e2, QName q2, Value v3, Child c2, Value v4 where d.docURIID=u.uriID and u.uriValID=v1.valID and v1.value="med.xml" and d.rootElemID=tc.parentID and tc.childID=e1.elID and e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and v2.value="entry" and c1.parentID=e1.elID and c1.childID=e2.elID and e2.elQNameID=q2.qnLocalID and q2.qnLocalID=v3.valID and v3.value="date" and e2.elID=c2.parentID and c2.childValID=v4.valID and v4.value="1/9/90"

Figure 5: Translation example of a simple FLWR expression.

rule TR_7 can be applied, with the following notations. Let E_0 be the part of the query providing bindings for the query variables $\overrightarrow{x} = x_1, \ldots, x_n$ (in the case of FLWR expressions, the *for-where* clauses); tuples resulting from $T(E_0)$ contain bindings for the variables in $\overrightarrow{x}$. We denote the tag of the outermost result element by $E_1(\overrightarrow{x})$. Let $E_2, \ldots, E_{2k}$ be the expressions providing names for the element's attributes, while $E_3, \ldots, E_{2k+1}$ provide attribute values. Let $H_1, \ldots, H_j$ be the expressions corresponding to the result element's children. Finally, let $G_1, \ldots, G_l$ be the elementary expressions (no element constructor) appearing in the E_is and H_is that really depend on the bound variables $\overrightarrow{x}$; each G_i provides values to be used as attribute or element names, attribute values, or character data. The first union term contains the translation of the *for-where* clause, padded with nulls; this term contains only the variable bindings, and is labeled 0. Each of the next l terms retrieves the information corresponding to one of the $G_1, \ldots, G_l$ path expressions.

TR$_7$ $T(< E_1(\overrightarrow{x}) \quad E_2(\overrightarrow{x}) = E_3(\overrightarrow{x}) \ldots$
$\ldots$
$E_{2k}(\overrightarrow{x}) = E_{2k+1}(\overrightarrow{x}) >$
$H_1(\overrightarrow{x})$
$\ldots$
$H_j(\overrightarrow{x})$
$</E_1(x)>) =$

with $T(E_0)$ as BoundVars
(select bv.*, 0 as label, null as $g_1, \ldots,$ null as g_l
from BoundVars bv
$\cup$
select bv.*, 1 as label, $S(G_1)$ as $g_1, \ldots,$ null as g_l
from BoundVars bv, $F(G_1)$
where $W(G_1)$
$\cup \ldots$
select bv.*, $l+1$ as label, null as $g_1, \ldots, S(G_l)$ as g_l
from BoundVars bv, $F(G_l)$
where $W(G_l)$)
order by bv.*, label

As a by-product of the translation from normalized XQuery to SQL, a tagging template is constructed, to inform the tagger module how to structure data from the sorted tuples into an XML result. As an example, consider the normalized query in figure 6, and its corresponding tagging template. Running this query on our medical database yields one binding for the variables $x_1, x_2, k = 0$ (no attributes in the returned element), $j = 2$, H_1 is the element constructor with tag *personal*, H_2 is the element constructor with tag *medical*; $l = 3$, G_1 is *\$x$_1$/name*, G_2 is *\$x$_1$/address*, G_3 is *\$x$_2$/entry*.

We briefly explain the construction of the tagging template, during the translation of a complex FLWR expression. First, we translate the simplified FLWR expression having the same for and where clauses as the

```
for $x1 in document("med.xml")//patient,
    $x2 in document("med.xml")//record
where $x1/@SSno=x2/patientSSno
return <medFile>
  <personal>
    <patName> $x1/name </patName>
    <patAddress> $x1/address </patAddress>
  </personal>
  <medical> $x2/entry </medical>
</medFile>

<template disc="label">
  <elem tag="medFile">
    <elem tag="personal">
      <elem tag="patName">
        <directContent col="g1"/>
      </elem>
      <elem tag="patAddress">
        <directContent col="g2"/>
      </elem>
    </elem>
  </elem>
</template>
```

Figure 6: Sample query and its tagging template.

complex expression, and returning only the bound variables: this yields the subquery $T(E_0)$ in rule TR_7 (an example for $T(E_0)$ is the SQL query in figure 5). Next, the structure of the returned element is copied into the tagging template as follows. Constants appearing in the result are copied as such in the template. Every G_i in the result yields: a new union term to the sorted union query, joining the result of the *for-where* block and the translation of G_i; and an *elem* entry in the template. This amounts to multiple outer joins between the bound variables and the expressions retrieving components of the result that depend on these variables.

Each block of the sorted union query will be rewritten and handed to the execution engine. The result metadata (column number, types and names) stay the same in the queries over the virtual and real schemas; therefore, the column information contained in the tagging template can be used by the tagger to structure the result. For every tuple labeled 0, the tagger starts a new element; then, by following the label field, it decides where to fill in the value from the non-null g_i column. The tagger runs in linear time and constant space [20].

5 Relational query rewriting

Until now, we have shown how to normalize XQuery queries, and how to translate them into SQL over the virtual schema, when the translation is possible. During normalization and translation, the local data sources are ignored, and all transformations are performed on the user query. This section describes the relational query rewriting phase, in which we finally connect the query to the data sources; the query is rewritten using the descriptions of local data sources as views over the generic relational schema.

We illustrate the rewriting process on the database shown in section 2.2: the data presented at the global level is contained in the **Med.XML** document, one local data source stores patient information in a Patient(name,dob,SSno,address) table, while medical records are stored as such in an XML document.

5.1 View definitions for relational sources

Figure 7 shows the view definition for the **Patient** table. This view relates the information in the table to data items from the **Med.XML** document. The first three tables in the from clause, and the first three predicates in the where, give the name of the document. The next few joins represent the information that the root element of the document, e1, has a *patient* tag, while the joins in line 8 of the view retrieve its *tuple* children. For each *tuple* element (e2 in the query), the *SSno* attribute of the element provides the SSno field in the Patient table (v5.value is the actual value to be found in the element). Lines 10-11, 12-13 and 14-15 have the same structure; they describe the *name*, *dob* and respectively *address* children of the tuple elements. Each of these three children (e3, e4 and e5) contains a value corresponding to a field in the Patient table's tuples; these values, v5, v7 and v9, appear in the project list. Besides the actual attribute values, this view also exports element IDs of all elements in the view definition; we have already discussed the need for IDs in real data collections in section 4.1.

5.2 View definitions for DOM sources

Agora is capable of processing both relational and DOM-compliant data sources. For example, to exploit a data source stored as an XML file, the DOM wrapper constructs a DOM representation of the file by invoking a parser. API calls on the resulting DOM tree can be used to access its content. For example, the call *x.getDescendents("someTag")*, where x is a node in the DOM tree (corresponding to an XML element) returns the list of x's descendents labeled *someTag*. This call is modeled as a three-attribute relation **Descendent(ancestor, descendent, tag)**; from the query engine's point of view, the DOM wrapper manages several tables, one per possible DOM API call. There is one subtlety regarding these tables: they have access restrictions, in the sense that their content cannot be scanned. The full extent of the **Descendent** table, for example, cannot be obtained: the only way to obtain tuples from this table is to supply a value for the **ancestor** field. In Agora, we model such restrictions by binding patterns [7].

The virtual tables exported by the DOM wrapper are described as views over the virtual generic relational

```
for x in document("med.xml")/medical/patient,
    y in document("med.xml")//patientSSno, z in x/name
where x/@SSno=y
return z

select e3.elID as $z
from Document d1, URI u1, Value v1, Element e1, QName q1,
     Value v2, Child c1, Element e2, QName q2, Value v3,
     Attribute a1, Value v4, Value v5, Child c2, Element e3,
     QName q3, Value v6, TransClosure tc1, Element e4,
     QName q4, Value v7, Child c3, Value v8
where d1.docURIID=u1.uriID and u1.uriValID=v1.valID and
     v1.value="med.xml" and d1.rootElemId=e1.elID and
     e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID
     and v2.value="medical" and c1.parentID=e1.elID and
     c1.childID=e2.elID and e2.elQameID=q2.qNameID and
     q2.qnLocalID=v3.valID and v3.value="patient" and
     a1.attrElID=e2.elID and a2.attrNameID=v4.valID and
     v4.value="SSno" and a1.attrValID=v5.valID and
     c2.parentID=e2.elID and c2.childID=e3.elID and
     e3.elQNameID=q3.qNameID and q3.qnLocalID=v6.valID
     and v6.value="name" and d1.rootElemId=tc2.parentID
     and tc2.childID=e4.elID and e4.elQameID=q4.qNameID
     and q4.qnLocalID=v7.valueID and
     v7.value="patientSSno" and c3.parentID=e3.elID and
     c3.childValID=v8.valID and v5.value=v8.value
```

Figure 8: XQuery query and its SQL translation.

schema, just like the tables from relational data sources. Here is the view definition corresponding to the **Descendent** table :

```
select tc.ancestor as anc, tc.descendent as desc, v.value as tag
from TransClosure tc, Element e, QName q, Value v
where tc.desc=e.elID and e.elQNameID=q.qNameID and
     qnLocalID=v.valID
```

5.3 Rewriting algorithm

Given the translated query and the view definitions, a query rewriting algorithm searching for maximally contained rewritings [13] is used to produce a query to be sent to the data sources. In a large-scale data integration application, such an algorithm is appropriate, since there is no guarantee that all qualifying data is available. In a different scenario, where one or a few relational sources are integrated under an XML global schema, a rewriting algorithm searching for equivalent query rewriting can be used, as we did in [15]. It is known that the problem of rewriting a query using a set of views is NP-hard, whether equivalent or maximally-contained rewritings are desired [13]. This complexity is the price to pay for the advantages of the LAV approach; however, recent work done in [16] for maximally contained rewritings and in [9] for equivalent rewritings presents efficient implementations that scale up well for large queries.

As an example, consider the rewriting of the query in figure 8, shown together with its translation on the generic schema. For each *patientSSno* element, the query returns the names of patients with a matching SSno attribute; on the sample document in figure 1, this query would return "Doe, John". The *record* elements of the **Med.XML** file are stored as such in an XML document, managed by a DOM wrapper as described above; thus, the query joins information from a relational table and from a native XML document. Here

```
select v7.value as name, v9.value as dob, v5.value as SSno, v11.value as address,
      e1.elID as e1, e2.elID as e2, e3.elID as e3, e4.elID as e4, e5.elID as e5, e6.elID as e6
from Document d1, URI u1, Value v1, Element e1, QName q1, Value v2, Child c1, Element e2, QName q2, Value v3,
      Attribute a1, Value v4, Value v5, Child c2, Element e3, QName q3, Value v6, Child c3, Value v7, Child c4
      Element e4, QName q4, Value v8, Child c5, Value v9, Child c6, Element e5, QName q5, Value v10, Child c7, Value v11
where d1.docURIID=u1.uriID and u1.uriValID=v1.valID and v1.value="med.xml" and d1.rootElemID=e1.elID and
      e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and v2.value="patient" and c1.parentID=e1.elID and
      c1.childID=e2.elID and e2.elQNameID=q2.qNameID and q2.qnLocalID=v3.valID and v3.value="tuple" and
      a1.attrElID=e2.elID and a1.attrNameID=v4.valID and v4.value="SSno" and a1.attrValID=v5.valID and
      c2.parentID=e2.elID and c2.childID=e3.elID and e3.elQNameID=q3.qNameID and q3.qnLocalID=v6.valID and
      v6.value="name" and c3.parentID=e3.elID and c3.childValID=v7.valID and
      c4.parentID=e2.elID and c4.childID=e4.elID and e4.elQNameID=q4.qNameID and q4.qnLocalID=v8.valID and
      v8.value="dob" and c5.parentID=e4.elID and c5.childValID=v9.valID and
      c6.parentID=e2.elID and c6.childID=e5.elID and e5.elQNameID=q5.qNameID and q5.qnLocalID=v10.valID and
      v10.value="address" and c7.parentID=e5.elID and c7.childValID=v11.valID
```

Figure 7: View definition for the Patient table .

is the rewritten SQL query resulting from our example:

```
select p.name as z
from REL:Patient p, DOM:Document d, DOM: Descendent desc
      DOM: Descendent desc
where d.docName="med.xml" and d.docRoot=desc.ancestor
      and desc.descendent=n.node and desc.tag="SSno"
```

The relational query rewriting algorithm uses the view definition for the Patient table from figure 7, and the set of view definitions corresponding to the virtual tables exported by the DOM wrapper. In the rewritten query, tables corresponding to local data sources are prefixed with the name of the wrapper managing them: REL for a relational wrapper, and DOM for the wrapper holding the XML document.

In this example, the query fragment corresponding to the *document("med.xml")/medical/patient* path expression (no // step) has been rewritten using the view for the Patient table, that describes the same path; for the fragment corresponding to *document("med.xml")//patientSSno*, a view definition using the TransClosure table has been identified. These are simple cases in which the query and the view correspond syntactically (either both use a recursive descent step or none of them uses it). However, syntactic correspondence is not required in order to use a view to answer a query; the SQL query rewriter encapsulates several types of semantic information. As a simple example, the rewriter is aware that a view defined with a Element-Child-Element join is contained in a query having a Element-TransClosure-Element join, simply because children are a subset of descendents. For more complex view-query combinations, the DTD of the global document is used to decide whether the view is a subset of the query or not. For example, if the query is *document("med.xml")/medical/records/SSno*, a view defined as *document("med.xml")//SSno* can be used only if the DTD implies that all SSno elements are on the path appearing in the query. In the absence of a DTD, the view cannot be used.

6 Translating queries with intermediate XML results

In this section, we explain how Agora's capacity to query native XML documents is used for translating queries necessitating the materialization of intermediate XML results.

```
for      $p in document("med.xml")//patient
where    $p/name="Doe, John"
return   closestHospitals($p/address, "hospitals.xml")
             //hospital[range 1 to 3]
```

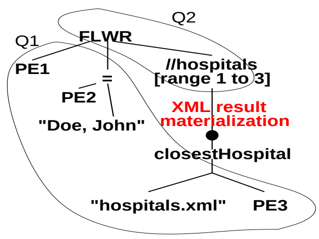

Figure 9: Query necessitating intermediate XML results materialization.

Consider the normalized query in figure 9, and its representation as an operator tree. In this tree, PE_1 corresponds to *document("med.xml")//patient*, PE_2 is *$p/name*, and PE_3 is *$p/address*; note that the bindings of *$p* from PE_1 need to be passed to PE_3. By examining a node, it can be decided whether (a) this expression cannot be executed by a relational processing system, and it is not a problem of intermediate XML results; in this case, the whole query is untranslatable; (b) this node does not necessitate materialization of its inputs; or (c) this node does necessitate the materialization of one or more of its inputs; in this case, XML materialization nodes are inserted in the query tree between the current node and its appropriate descendents. In this example, there is one such materialization node, as input to the *range* operator.

At this point, the input query is partitioned into two subqueries. Q_1 extracts the patient and passes the proper binding for *$p* as input to the function, which results in an XML document. This document is assigned to a DOM wrapper described in section 5.2, as a special temporary data source, given a new name, and provided as input to Q_2. Next, the subqueries are sorted in the order dictated by the data sources they produce/consume; Q_1, then Q_2, are translated into SQL queries on the generic schema, rewritten and executed. The fact that one input is a temporary document does not hinder Q_2's rewriting, since the DOM wrapper publishes *generic* view definitions, in which the XML document name is a simple attribute and can be selected on.

7 Related work

Projects like Garlic [17], Disco [21], Tsimmis [8] and Yat [2] all adopt the GAV approach, and therefore do not compare directly to our system. The Information Manifold [11] is the single data integration system with a LAV architecture; however, the local and global schemas are relational.

SilkRoute [4, 5] and XPERANTO [20, 19] focus on exporting relational databases under an XML interface. Since the mapping is done from tuples to XML, these projects adopt the GAV approach; also, they can only integrate relational data sources. In a work developed in parallel with ours, a translation methodology from XQuery to SQL is provided, in order to query XML views of relational data [19]. In contrast, our integration approach can handle diverse data sources, not only relational. The study in [4] investigates efficient ways of materializing a large XML document from the data contained in an RDBMS. In this context, a single sorted outer union SQL query may be suboptimal, and the authors describe a search space of several smaller SQL queries. We used the sorted outer union approach for several reasons. First, we expect that in a data integration setting, most queries return moderate-size results. Also, the search done in [4] is based on a RDBMS's optimizer's cost estimates for a given SQL query; in a centralized context, these estimates are easy to obtain. However, in a data integration context, it is difficult to get precise and *comparable* estimates from wrappers.

Work done in [3, 18, 6] investigated ways of storing XML documents in tables. Our approach can handle all the mappings they produce, since the relational storage is defined as materialized views over the XML documents. In [3], "lossy" mappings (that do not store all data in a document) are forbidden, while we allow any mapping; also, the query language they use does not construct new XML structure, while XQuery does.

8 Conclusion

We have presented a methodology for integrating relational and tree-structured data sources, in particular XML documents, under a single XML global schema; our work is the first solution to this problem using the LAV approach, which is preferable for large-scale data integration applications. We isolated the syntactical translation step (from the users' XML query into a SQL query on a generic schema) from the semantic step, which identifies the relevant data sources to answer the query. Our approach is implemented into the Agora research prototype, and we measured reasonable performances for the relational (equivalent) rewriting algorithm: less than 1 sec. for a query over 25 tables, 3 documents, using 10 views (all Agora is implemented in Java, we used JDK 1.3 on a Pentium 233, running RedHat Linux 6.2) [15]. In the future, we plan to study, in a relational-only integration context (all data sources stored into RDBMSs), the translation of XQuery queries with updates (to be standardized soon) into SQL queries over the local sources.

Acknowledgements We thank Leonid Libkin and Jim Melton for our interesting discussions on the expressive power of SQL, and Alberto Lerner for his careful proofreading and helpful comments.

References

[1] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.

[2] Vassilis Cristophides, Sophie Cluet, and Jérôme Siméon. On wrapping query languages and efficient XML integration. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2000.

[3] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 431–442, 1999.

[4] Mary Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middleware queries. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2001.

[5] Mary Fernandez, Wang-Chiew Tan, and Dan Suciu. Silkroute: Trading between relational and XML. In *Proc. of the Int. WWW Conf.*, May 2000.

[6] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. In *IEEE Data Engineering Bulletin*, volume 22(3), pages 27–34, 1999.

[7] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 311–322, 1999.

[8] H. Garcia-Molina, Yannis Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[9] J. Goldstein and P. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2001.

[10] Alon Halevy. Logic-based techniques in data integration. *Logic Based Artificial Intelligence*, 2000.

[11] T. Kirk, Alon Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*, 1995.

[12] `http://www-caravel.inria.fr/LeSelect`.

[13] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views.

In *Proc. of the Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[14] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Pushing XML queries inside relational databases. Tech. Report no. 4112, INRIA. Available at `www-caravel.inria.fr/Epublications.html`, 2001.

[15] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Dan Olteanu, and Florian Xhumari. Agora: Living with XML and relational (system demonstration). In *Proc. of the VLDB Conf.*, 2000.

[16] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In *Proc. of the VLDB Conf.*, 2000.

[17] Mary Tork Roth and Peter M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the VLDB Conf.*, pages 266–275, 1997.

[18] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the VLDB Conf.*, 1999.

[19] J. Shanmungasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proc. of the VLDB Conf.*, 2001.

[20] J. Shanmungasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proc. of the VLDB Conf.*, Cairo, Egypt, 2000.

[21] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to distributed heterogeneous data sources with Disco. IEEE Transactions On Knowledge and Data Engineering, 1998.

[22] XQuery 1.0 formal semantics. `http://www.w3.org/TR/query-algebra`, 2001. Work in progress.

[23] Document Object Model. `http://www.w3.org/DOM/`, 1998.

[24] XQuery: A query language for XML. `http://www.w3.org/TR/xquery`, 2001. Work in progress.