SilkRoute: Trading between Relations and XML

Mary Fernández	AT&T Labs – Research	mff@research.att.com
Wang-Chiew Tan	University of Pennsylvania	wctan@cis.saul.upenn.edu
Dan Suciu	AT&T Labs – Research	suciu@research.att.com

November 22, 1999

Abstract

Keywords. Data exchange, XML, Relational databases, XML Queries.

XML is the standard format for data exchange between inter-enterprise applications on the Internet. To facilitate data exchange, industry groups define public document type definitions (DTDs) that specify the format of the XML data to be exchanged between their applications. In this paper, we address the problem of automating the conversion of relational data into XML. We describe SilkRoute, a *general, dynamic*, and *efficient* tool for viewing and querying relational data in XML. SilkRoute is general, because it can express mappings of relational data into XML that conforms to arbitrary DTDs, not just a canonical mapping of the relational schema. We call these mappings *views*. Applications express the data they need as an XML-QL query over the view. SilkRoute is dynamic, because it only materializes the fragment of an XML view needed by an application, and it is efficient, because it fully exploits the underlying RDBMs query engine whenever data items in an XML view need to be materialized.

1 Introduction

XML can serve many purposes: as a more expressive markup language than HTML; as a object-serialization format for distributed object applications; or as a data exchange format. In this work, we focus on the role of XML in data exchange, in which XML documents are generated from persistent data then sent over a network to an application. Numerous industry groups, including health care and telecommunications, are working on document type definitions (DTDs) that specify the format of the XML data to be exchanged between their applications[1]. The aim is to use XML as a "lingua franca" for data exchange between inter-enterprise applications, making it possible for data to be exchanged regardless of the platform on which it is stored or the data model in which it is represented.

In this paper, we address the problem of exporting existing data into XML. Most data is stored in relational or object-relational database management systems (RDBMs) or in legacy formats. To realize the full potential of XML, we need tools that can automatically convert the vast stores of relational data into XML: we call the resulting XML document a *view*. We believe such tools must be *general*, *dynamic*, and *efficient*. Generality, in particular, is a hard requirement, which is not currently met either by commercial products or by research prototypes Relational data is tabular, flat, normalized, and its schema is proprietary, which makes it unsuitable for direct exchange. In contrast, XML data is nested and unnormalized, and its DTD is public. Thus, the mapping from the relational data to the XML view is often complex, and a conversion tool must be *general* enough to express complex mappings. Existing systems are not general, because they map each relational database schema into a fixed, canonical DTD. This approach is limited, because no public DTD will match exactly a proprietary relational schema. In addition, one may want to map one relational source into multiple XML documents, each of which conforms to a different DTD.

[1]Dozens of such applications can be found at `http://www.oasis-open.org/cover/`.

Our second requirement is that tools must be dynamic, i.e., only the fragment of the XML document needed by the application should be materialized. In database terminology, the XML view must be *virtual.* The application typically specifies in a query what data item(s) it needs from the XML document, and these items are typically a small fraction of the entire data. In commercial database systems, one can write a script to export relational data into XML documents. By our definition, these tools are general but not dynamic, because the entire document is generated all at once.

Finally, to be efficient, such tools must exploit fully the underlying RDBMs query engine whenever data items in the XML view need to be materialized. Query processors for native XML data are still immature and do not have the performance of highly optimized RDBMs engines.

In this paper, we describe SilkRoute, a general, dynamic, and efficient tool for viewing and querying relational data in XML. In SilkRoute, data is exported into XML in two steps. First, an XML view of the relational database is defined using a declarative query language, called RXL (Relational to XML Transformation Language). The resulting XML view is virtual. Second, some other application formulates a query over the virtual view, extracting some piece of XML data. For this purpose, we use an existing XML query language, XML-QL. Only the result of that XML-QL query is materialized.

The core of SilkRoute is RXL, a powerful, declarative data transformation language from flat relations to XML data. On the relational side, RXL has the full power of SQL queries and can express joins, selection conditions, aggregates, and nested queries. On the XML side, RXL has the full power of XML-QL, and can generate XML data with complex structure with arbitrary levels of nesting. It can also specify arbitrary grouping criteria, using nested queries and Skolem functions. Typical RXL queries are long and complex, because they express general transformations from the relational store to the XML view. RXL has a block structure to help users organize, structure, and maintain large queries.

Once the virtual XML view is defined, SilkRoute accepts XML-QL user queries and *composes* them automatically with the RXL query. The result of the composition is another RXL query, which extracts only that fragment of the relational data that the user requested. Query composition is the most important technical contribution of this work, and our solution to this problem is general enough to be used in other systems requiring virtual XML views. In relational databases, composition is straightforward and not considered a problem; for example, Ramakrishnan [18] describes how to reformulate SQL queries over SQL virtual views as SQL queries over base relations. For XML, however, this problem is more complex, and, to the best of our knowledge, no solution has been published. We present here a sound, complete, and conceptually simple algorithm that when given an RXL query and an XML-QL query, produces a new RXL query equivalent to their composition. We place some restrictions on aggregate functions in RXL queries, but they can be used freely in XML-QL queries.

Finally, when an RXL query is evaluated, most of the processing is done by the underlying relational engine. To do this, the RXL query is split into a collection of SQL queries, each of which produces a set of tuples. The SQL queries are sent to the RDBMS, and their flat, sorted results are merged in a single pass to construct the nested XML output.

In summary, this paper makes the following contributions:

- We describe a general framework for mapping relational databases to XML views, to be used in data exchange.
- We describe a new query language, RXL, for mapping relational sources to XML views.
- We describe a sound and complete query composition algorithm that, when given an RXL query and an XML-QL query, generates a new RXL query equivalent to their composition.
- We describe a technique by which most of the work in an RXL query can be shipped to the underlying database engine.

To motivate this work, we present in the next section an example scenario from electronic commerce. In Section 2, we describe SilkRoute's architecture and view-definition language, RXL. Section 2.2 briefly

```
<?xml encoding="US-ASCII"?>
<!ELEMENT supplier (company, product*)>
<!ELEMENT product  (name, category, description, retail, sale?, reports*)>
<!ATTLIST product  ID ID>
<!ELEMENT company  (#PCDATA)>
<!ELEMENT name     (#PCDATA)>
<!ELEMENT retail   (#PCDATA)>
<!ELEMENT sale     (#PCDATA)>
<!ELEMENT report   (#PCDATA)>
<!ATTLIST report code (size|defective|style) #REQUIRED>
```

Figure 1: `supplier.dtd` : DTD of XML data exported by suppliers to resellers.

```
Clothing(*pid, item, category, description, price, cost)
SalePrice(*pid, price)
Problems(pid, code, comments)
```

Figure 2: Schema of supplier's relational database. (* denotes key).

describes XML-QL. In Section 3, we give an informal description of query composition and also the complete query composition algorithm. A brief discussion of aggregates appears in Section 3.3. SilkRoute is a particular instance of a mediator system, as defined by Geo Wiederhold [20]. Section 4 includes descriptions of related systems.

1.1 Motivating Example

We motivate SilkRoute with a simple example, from electronic commerce, in which *suppliers* provide product information to *resellers*. For their mutual benefit, suppliers and resellers have agreed to exchange data in a format that conforms to a particular DTD, depicted in Figure 1. It includes the supplier's name and a list of available products. Each product element includes an item name, a category name, a brief description, a retail price, an optional sale price, and zero or more trouble reports. The contents of a `retail` or `sale` element is a currency value. A trouble report includes a code attribute, indicating the class of problem; the report's contents are the customer's comments. Most importantly, this DTD is used by suppliers and resellers, and it is a public document.

Consider now a particular supplier whose business data is organized according to the relational schema depicted in Fig. 2. The `Clothing` table contains tuples with a product id (the table's key), an item name, category name, item description, price, and cost. The `SalePrice` table contains sale prices and has key field `pid`; the `Problem` table contains trouble codes and reports. This is a third-normal form relational schema, designed for the supplier's particular business needs. The schema is proprietary: for example, the supplier may not want to reveal the attribute `cost` in `Clothing`. The supplier's task is to convert its relational data into a valid XML view conforming to the DTD in Fig. 1 and make the XML view available to resellers. In this example, we assume the supplier exports a subset of its inventory, in particular, its stock of winter outer-wear that it wants to sell at a reduced price at the end of the winter season.

Once the XML views of suppliers' data are available, the reseller can access that data by formulating queries over the XML view. Some examples of such queries are:

- Retrieve products whose sale price is less than 50% of retail price.
- Count the number of "defective" reports for a product.

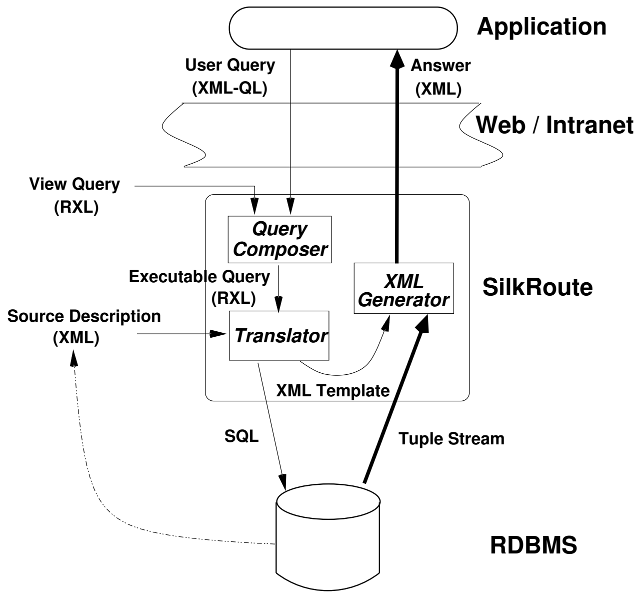

Figure 3: SilkRoute's Architecture.

- Compute minimum and maximum cost of outer-wear stock.

As these queries suggest, the reseller is typically only interested in a small subset of the information provided by the suppliers. Readers familiar with SQL will recognize that these queries could be formulated as SQL queries over the supplier's relational database, but relational schemas differ from supplier to supplier and are not accessible by the reseller.

2 SilkRoute's Architecture

SilkRoute's architecture is depicted in Figure 3. It serves as middleware between a relational database (RDBMS) and an application accessing that data over the Web.

The database administrator starts by writing an RXL query that defines the XML virtual view of the database. This is called the *view query* and is typically complex, because it transforms the relational data into a deeply nested XML view. The resulting view query is virtual, meaning that it is not evaluated, but kept in source code.

Typically, applications contact SilkRoute to request data. An application only "sees" the virtual XML view, not the underlying relational database. To access the data, it formulates in XML-QL a *user query* over the virtual view and sends it to SilkRoute. Together, the RXL view query and the XML-QL user query are passed to the *query composer*, the most complex module in SilkRoute. The composer computes the composition and produces a new RXL query, called the *executable query*. The answer to the executable query typically includes only a small fragment of the database, e.g., one data item, a small set of data items, or an aggregate value. Its result is an XML document, as specified by the user XML-QL query.

Once computed, the executable query is passed to the *translator*, which partitions it into a data extraction part, i.e., one or more SQL queries, and an XML construction part, i.e., an XML template. The translator also takes as input a description of the relational schema.

Until now, SilkRoute has manipulated only query source code, no data. At this point, the SQL queries are sent to the RDBMS server, which returns one tuple stream for each SQL query. The *XML generator* module merges these tuple streams and produces the XML document, which is then returned to the application.

This scenario is probably the most common use of SilkRoute, but minor changes to the information flow in Fig. 3 permit other scenarios. For example, the data administrator may export the entire database as one, large XML document by materializing the view query. This can be done by passing the view query directly to the translator. In another scenario, the result of query composition could be kept virtual for later composition with other user queries. This is useful, for example, when one wants to define a new XML view from an existing composed view. In the rest of this section, we describe the system's components in more detail.

2.1 The View Query: RXL

In this section, we describe RXL (Relational to XML transformation Language). RXL essentially combines the extraction part of SQL, i.e., a `from` and a `where` clause (possibly followed by `sort by` and/or `group by` clauses) with the construction part of XML-QL [9], i.e., the `construct` clause.

As a first example, consider this RXL query, which defines a fragment an XML view:

```
from  Clothing $c
where $c.category = "outerwear"
construct  <product>
              <name>$c.item</name>
              <category>$c.category</category>
              <retail>$c.price</retail>
           </product>
```

Given a database like that in Fig. 2, the query will produce an XML fragment like the following:

```
<product> <name>...</name> <category>...</category> <retail>...</retail> </product>
<product> <name>...</name> <category>...</category> <retail>...</retail> </product>
...
```

A root element is missing; we will explain later how to add one.

As in SQL, the `from` clause declares *variables* that iterate over tables. Variable names start with a `$`. In this example, `$c` is a tuple variable that iterates over the `Clothing` table. The `where` clause contains zero or more *filters* (boolean predicates) over *column expressions*. The column expression `$c.item` refers to the `item` attribute value of `$c` and in this case, requires that it equal the string `"outerwear"`. The `construct` clause specifies the XML value, called an *XML template*, in terms of the bound table variables.

RXL has three powerful features that make it possible to create arbitrarily complex XML structures: nested queries, Skolem functions, and block structure. An example of a *nested query* is:

```
from  Clothing $c
construct
  <product>
    <name>$c.item</name>
    { from Problems $p
      where $p.pid = $c.cid
      construct <report>$p.comments</report>
    }
  </product>
```

In each `<product>` element, the nested sub-query creates zero or more `<report>` sub-elements, one for each report associated to that product. Readers familiar with SQL may recognize this as a left outer join of `Clothing` with `Problems`.

Skolem functions allow us to control the way elements are grouped. Recall that in XML an attribute with type ID contains a value that uniquely identifies the element in the document, i.e., a *key*. In RXL, the distinguished attribute ID always has type ID, and its value is a Skolem term, which is used to control grouping and element creation. For example, in the following:

```
from  Clothing $c
construct
  <category ID=Cat($c.category) name=$c.category>
    <product>$c.item</product>
  </catebory>
```

Cat is a Skolem function and Cat($c.category) is a Skolem term whose meaning is that a new <category> element exists for every value of $c.category. Without the ID attribute and its Skolem term, the query would create one <category> element for each row in Clothing:

```
<category><product>p1</product></category>
<category><product>p2</product></category> ...
```

With the Skolem term, only one <category> element exists for each category name, and it includes all products in that category:

```
<category><product>p1</product><product>p2<product></category>
<category><product>p3</product><product>p4</product><product>p5</product></category>
...
```

When Skolem terms are missing, RXL introduces them automatically. Since Skolem terms could be used to define arbitrary graphs, RXL enforces semantic constraints that guarantee a view always defines a tree, and therefore, a well-formed XML document. For example, the Skolem term of a sub-element must include all the variables of its the parent element.

Finally, the *block structure* allows RXL to construct parts of complex elements independently. The following query has two blocks:

```
{ from  Clothing $c
  construct <product ID=Prod($c.name)>
              <name>$c.item</name>
              <price>$c.price</price>
            </product>   }
{ from  Clearance $d
  construct <product ID=Prod($d.prodname)>
              <name>$d.prodname</name>
              <discount>$d.disc</discount>
            </product>   }
```

The first block creates elements of the form:

```
<product><name>n</name><price>p</price></product>
```

for each product name in Clothing. The second block creates elements of the form:

```
<product><name>n</name><discount>d</discount></product>
```

for each product name in Clearance. When the same product name occurs both in Clothing and Clearance, then the two elements will have the same ID key and are merged into:

```
<product> <name>n</name><price>p</price><discount>d</discount></product>
```

```
 1. construct
 2. <supplier ID=Supp()>
 3.   <company ID=Comp()>"Acme Clothing"</company>
 4.   {
 5.    from  Clothing $c
 6.     where $c.category = "outerwear"
 7.     construct
 8.       <product ID=Prod($c.pid)>
 9.         <name  ID=Name($c.pid,$c.item)>$c.item</name>
10.          <category ID=Cat($c.pid,$c.category)>$c.category</category>
11.          <retail ID=Retail($c.pid,$c.price)>$c.price</retail>
12.        { from SalePrice $s
13.          where $s.pid = $c.pid
14.          construct
15.          <sale ID=Sale($c.pid,$s.pid,$s.price)>$s.price</retail>
16.        }
17.        { from Problems $p
18.          where $p.pid = $c.pid
19.          construct
20.          <report code=$p.code ID=Prob($c.pid,$p.pid,$p.code,$p.comments)>
21.            $p.comments
22.          </report>
23.        }
24.        </product>
25.   }
26.  </supplier>
```

Figure 4: RXL view query (V)

Readers familiar with SQL will recognize this as an outer join.

Figure 4 contains the complete view query for our supplier example in Sec. 1.1. Lines 1, 2, and 26 create the root `<supplier>` element : notice that the Skolem term `Supp()` has no variables, meaning that one `<supplier>` element is created. The outer-most clause constructs the top-level element `supplier` and its `company` child element. The first nested clause (lines 4–26) contains the query fragment described above, which constructs one `product` element for each "outerwear" item. Within this clause, the nested clause (lines 12–16) expresses a join between the `Clothing` and `SalePrice` tables and constructs a `sale` element with the product's sale price nested within the outer `product` element. The last nested clause (lines 17–23) expresses a join between the `Clothing` and `Problem` tables and constructs one `report` element containing the problem code and customer's comments; the `report` elements are also nested within the outer `product` element. Notice that the Skolem term of `product` guarantee that all `product` elements with the same identifier are grouped together. Usually Skolem terms are inferred automatically, but we include them explicitly, because they are relevant to query composition described in Sec. 3.

2.2 The User Query: XML-QL

Applications do not access the relational data directly, but through the XML view. To do this, they write *user queries* in XML-QL, a query language specifically designed for XML [9]. XML-QL queries contain a `where` clause followed by a `construct` clause. The `where` clause contains an arbitrary number of XML *patterns* and filters. The `construct` clause is identical to that in RXL.

```
 1. construct
 2.  <results>
 3.  { where <supplier>
 4.            <company>$company</company>
 5.            <product>
 6.               <name>$name</name>
 7.               <retail>$retail</retail>
 8.               <sale>$sale</sale>
 9.            </product>
10.          </supplier> in "http://acme.com/products.xml",
11.          $sale < 0.5 * $retail
12.    construct
13.          <result ID=Result($company)>
14.            <supplier>$company</supplier>
15.            <name>$name</name>
16.          </result>
17.  }
18.  </results>
```

Figure 5: XML-QL user query (U).

In our example, the reseller can retrieve all products with sale price less than half of retail price using the XML-QL query in Figure 5. The `where` clause consists of a pattern (lines 3-10) and a filter (line 11). A pattern has the same syntax as XML data, but also may contain variables, whose names start with `$`. Filters are similar to RXL (and SQL). The meaning of a query is as follows. First, all variables in the `where` clause are bound in all possible ways to the contents of elements in the XML document. For each such binding, the `construct` clause constructs an XML value. Grouping is expressed by Skolem terms in the `construct` clause. In this example, the construct clause produces one `result` element for each value of `$company`; each `result` element contains the supplier's name and a list of `name` elements containing the product names.

In this example, it is important to notice that answer to the user query includes a small fraction of the relational database, i.e., only those products that are heavily discounted.

2.3 The Query Composer

SilkRoute's query composer takes a user query and the RXL view query and generates a new RXL query, which is equivalent to the user query evaluated on the materialized view. In our example, the view query is in Fig. 4, the user query in Fig. 5, and the composed query is in Fig. 6. The composed query combines Those fragments from the user query are highlighted. The composed query extracts data from the relational database in the same way as the view query. It also includes the user filter `$s.price < 0.5 $c.retail` and structures the result as in the user query. The details of composition are subtle, and a complete description of the composition algorithm is given in Section 3.

We call the composed query *executable*, because it is typically translated into SQL queries and sent to the relational database engine. Notice that the answer of the executable query is quite small — the same as that of the user query. In general, it is more efficient to execute the composed query, instead of materializing the view query, because composed queries often contain constraints on scalar values that can be evaluated using indexes in the relational database. Such indices are of little or no use when evaluating a view query. For example, consider a user query that specifies the condition: `$s.price between 80 and 100`. This condition is propagated into the executable query, and then into the SQL query, and can be evaluated efficiently if an index exists on `price`. In contrast, an index on `price` is useless when materializing the view query directly.

```
construct
  <results>
  { from  Clothing $c, SalePrice $s
    where $c.category = "outerwear",
          $c.pid = $s.pid,
          $s.price < 0.5 * $c.retail
    construct
      <result ID=Result("Acme Clothing")>
        <supplier>"Acme Clothing"</supplier>
        <name ID=Name($c.pid, $c.item)>$c.item</name>
      </result>
  }
  </results>
```

Figure 6: Composed RXL query (C).

2.4 Translator and XML Generator

The *translator* takes an RXL query and decomposes it into one or more SQL queries and an XML template. The SQL queries are executed by the relational engine, and their flat results (streams of tuples) are converted into XML by the *XML generator*.

The translator also takes a source description, which is an XML document specifying systems information needed to contact the source: the protocol (e.g. `JDBC`), the connection string, and a source-specific query driver. The driver translates RXL expressions into the source's query language, which is typically a dialect of SQL, but other query languages can be supported. For example, the executable RXL query in Fig. 6 is translated into the SQL query:

```
select  c.pid as pid, c.item as item
from    Clothing c, SalePrice s
where   c.category = "outerwar",
        c.pid = s.pid,
        s.price < 0.5 * c.retail
sort by c.pid
```

and into the XML template:

```
 <result ID=Result("Acme Clothing")>
      <supplier> "Acme Clothing" </supplier>
      <name ID=Name($pid, $item)>$item</name>
 </result>
```

where the variables `$pid` and `$item` refer to the attributes `pid` and `item` in the SQL query's `select` clause. After translation, the SQL query is sent to the relational engine, and the resulting tuple stream is fed into the XML generator, which produces the XML output.

In this example, the translation requires only one SQL query. In general, there may be several ways to translate a complex RXL query into one or more SQL queries and to merge tuple streams into the XML result. Choosing an efficient evaluation strategy may be important when the RXL query returns a large result, e.g., if the entire XML view if materialized. Currently, SilkRoute has one evaluation strategy, which generates one SQL query for each RXL sub-query, which must be in conjunctive-normal form (CNF). Each SQL query has a `sort by` clause, making it possible for the XML generator to merge them into an XML document in a single pass.

2.5 Alternative Approaches

We have described what we believe to be the most general approach for exporting relational data into XML. Other approaches are possible, and in some cases, may be more desirable.

Currently, the standard interface to a relational database is an HTML form, which constructs a relational query from user inputs. Typically, such interfaces render the results of a query in HTML, but they could generate XML instead. Forms interfaces are appropriate for casual users, but inappropriate for data exchange between applications, because they limit the application to only those queries that are predetermined by the form interface. Aggregate queries, for example, are rarely offered by form interfaces.

In another alternative, the data provider can either precompute the materialized view or compute it on demand whenever an application requests it. This is feasible when the XML view is small and the application needs to load the entire XML view in memory, e.g., using the DOM interface. However, precomputed views are not dynamic, i.e., their data can become stale, and are not acceptable when data freshness is critical.

A third alternative is to use a native XML database engine, which can store XML data and process queries in some XML query language. It is unlikely XML engines could replace completely high-performance relational databases, but a high-performance XML engine might be appropriate for data exchange. For example, one could materialize an XML view using SilkRoute and store the result in an XML engine, which would process subsequent XML-QL queries directly on the stored, materialized view. Given an efficient XML engine, this scenario might be better than our solution in SilkRoute, because it avoids the cost of query composition. We don't expect, however, that native implementations of XML-QL, or any other XML query language, on XML documents will attain performance comparable to that of SQL engines on relational data anytime soon. In addition, this approach does not guarantee data freshness and incurs a high space cost by materializing the data completely in XML.

3 Query Composition

In this section, we give an informal description of query composition, using the example view in Fig 4 and the query in Fig. 5, and then a detailed description of the query composition algorithm.

The problem statement is the following. We are given as inputs the source code of an RXL query V and an XML-QL query U. We have to construct the source code of an RXL query C, whose meaning is the composition of V and U: $C = U \circ V$. In our example, V is the view definition (Fig. 4), and U is the user query (Fig. 5): the composed query, C, is given in Fig. 6. Specifically, the RXL query V takes as input a relational database instance RDB and returns an XML document XD: $XD = V(RDB)$. The XML-QL query U takes as input the XML document and returns an answer, which is just another XML document, $A = U(XD) = U(V(RDB))$. We have to construct some RXL query C which, when given RDB, returns A directly, i.e., $C(RDB) = A = U(V(RDB)) = (U \circ V)(RDB)$.

The intuition behind the composition algorithm is simple. The key observation is that all XML tags, attributes, and `#PCDATA` present in the XML document XD occur explicitly in the `construct` clause(s) of V. However, U queries the document XD by matching a pattern against tags, attributes, and `#PCDATA` present in XD. The key idea is to match U's pattern on V directly, without constructing XD. In our example, U's patterns mentions `<supplier>`, `<company>`, `<product>`, `<name>`, `<retail>`, and `<sale>` with a particular nesting, and all these tags occur in V with the same nesting. The first step in the composition algorithm matches U's patterns with V's template. We show V again in Fig 7, after the matching, with the matched tags highlighted (in blue). The second step constructs the composed query C as follows. C's `construct` clause is the same as U's `construct` clause, modulo variable substitutions. Its `from` and `where` clauses consist both of the "relevant" `from` and `where` clauses in V and of all the `where` filter conditions in U, modulo variable renaming. This completes the construction of C. To illustrate the second step, consider again V with the constructors highlighted in Fig. 7: all their corresponding `from` and `where` clauses are "relevant" and, combined, they become:

```
from Clothing $c, SalePrice $s
```

```
construct
 <supplier ID=Supp()>
   <company ID=Comp()>"Acme Clothing"</company>
   {
     from  Clothing $c
     where $c.category = "outerwear"
     construct
       <product ID=Prod($c.pid)>
       <name  ID=Name($c.pid,$c.item)>$c.item</name>
       <category ID=Cat($c.pid,$c.category)>$c.category</category>
       <retail ID=Retail($c.pid,$c.price)>$c.price</retail>
        { from SalePrice $s
           where $s.pid = $c.pid
           construct
             <sale ID=Sale($c.pid,$s.pid,$s.price)>$s.price</sale>
        }
        { from Problems $p
           where $p.pid = $c.pid
           construct
           <report code=$p.code ID=Prob($c.pid,$p.pid,$p.code,$p.comments)>
             $p.comments
           </report>
        }
       </product>
   }
  </supplier>
```

Figure 7: RXL view query (V) with patterns from XML-QL query highlighted.

```
where $c.category = "outerwear", $s.pid = $c.pid
```

The query U has a single filter condition in the `where` clause, `$sale < 0.5 * $retail` which, after variable renaming becomes the RXL `where` filter:

```
where $s.price < 0.5 * $c.retail
```

The reader may check that these are indeed C's `from` and `where` conditions given in Fig. 6.

Figure 8 depicts the architecture of query composition. The query composer decomposes an XML-QL query (U) into patterns, filters, and a constructor and converts the RXL view (V) into an internal representation, called a view tree. The *pattern matcher* evaluates the XML-QL patterns on the view tree and produces a *solutions relation*, R. Each tuple in R represents one match of the patterns over the view tree and, therefore, one potential solution for the XML-QL query. Multiple matches may occur if the patterns contain alternation, e.g., `<company|organization>`, or Kleene-star operators, e.g., `<*.supplier>`.

The *rewriter* takes R, U's filters and `construct` clause, and V's `from` and `where` and rewrites each solution tuple into one RXL clause. The result is a new RXL view (C), which is equivalent to the XML-QL query applied to the original view. This technique is an example of *partial evaluation*: the patterns are evaluated at *composition time* (a.k.a. compile) on the view tree, and the filters and constructors are evaluated *at run time* when the new RXL view is evaluated. The remainder of this section describes the internal representation of view and user queries and gives a complete, pseudo-code description of the composition algorithm.

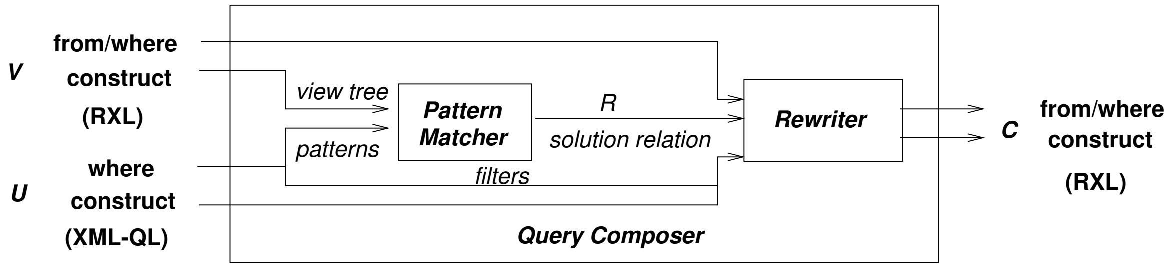

Figure 8: Diagram of Query Composition.

```
Root() =
<supplier ID=Supp()>
  <company ID=Comp()>"Acme Clothing"</company>
     <product ID=Prod(cpid)>
        <name ID=Name(cpid,citem)>citem</name>
        <category ID=Cat(cpid,ccategory)>
          ccategory
        </category>
        <retail ID=Retail(cpid,cprice)>
          cprice
        </retail>
        <sale ID=Sale(cpid,spid,sprice)>
          sprice
        </sale
        <report code=pcode ID=Prob(cpid,ppid,pcode,pcmnts)>
          pcmnts
        </report>
      </product>
</supplier>
```

Figure 9: View tree for RXL view in Figure 4.

3.1 View Tree

A view tree is an abstract representation of the XML document defined by an RXL view and is the *least upper bound* for that document, i.e., it contains exactly one instance of each element that could occur in the materialized document. Figure 9 contains the view tree for the RXL view in Figure 4. A view tree always has a distinguished root labeled `Root()`. Elements in the tree are labeled by the expressions which guard their creation. These expressions are represented by datalog rules: the rule's head is the Skolem term that creates the element and the rule's body contains the table expressions and filters that guard the element's creation. To simplify the figure, an element is labeled with the head of its associated rule. For example, the unique `supplier` element is guarded by the rule `Supp() :- true`, which is always true, because no predicate expression guards the element's creation. The `retail` elements, however, are guarded by the rule

```
Retail(cpid, cprice) :- Clothing(cpid, _, category, _, cprice), category = "outerwear"
```

which means that one `retail` element is created for each value of `cpid` and `cprice` that satisfies the table expression on the right-hand side. Figure 10 contains all the datalog rules associated with the elements in the view tree. In RXL's concrete syntax, it is convenient to associate one rule with the construction of multiple elements. In this representation, however, each element in the view is guarded by a distinct rule.

```
Root()                          :- true
Supp()                          :- true
Comp("Acme Supplier")           :- true
Prod(cpid)                      :- Clothing(cpid, _, category, _, _), category = "outerwear"
Name(cpid, citem)               :- Clothing(pid, citem, category, _, _), category = "outerwear"
Cat(cpid, ccategory)            :- Clothing(cpid, _, category, _, _), category = "outerwear"
Retail(cpid, cprice)            :- Clothing(cpid, _, category, _, cprice), category = "outerwear"
Sale(cpid, spid, sprice)        :- Clothing(cpid, _, category, _, _), category = "outerwear"
                                   SalePrice(spid, sprice), cpid = spid
Rep(cpid, ppid, pcode, pcmnts)  :- Clothing(cpid, _, category, _, _), category = "outerwear",
                                   Problems(ppid, pcode, pcmnts), cpid = ppid
```

Figure 10: Datalog rules for view tree in Figure 9.

In the formal description of the algorithm, we need a notation for describing the types of values, e.g., view trees, XML-QL clauses, that are manipulated. We denote types by grammar rules, like the following:

```
Node    :- Tag, Rule, [ Node ]
Rule    :- SkolemTerm, [ Body ]
Body    :- TableExpr(String, [ String ])
         | Filter(BoolExpr)
         | Or([ Body ], [ Body ])
```

These rules specify that a view tree `Node` is composed of a tag, a rule, and a list of children nodes; a `Rule` is composed of a Skolem term (its head) and a conjunctive list of body expressions; and a rule `Body` is either a table expression, a filter expression, or the disjunction of two lists of conjuncts.

An XMLQL clause is represented by a list of patterns, a list of filters, and a constructor, and an RXL clause by a list of rule bodies and a constructor:

```
XMLQL   :- [ Pattern ], [ Filter ], Constructor
RXL     :- [ Body ], Constructor
```

A constructor is either: a constant string; a variable; an element, which includes a tag and list of nested constructors; or a nested query. To simplify presentation, constructors are polymorphic, i.e., an XMLQL constructor contains only a nested XMLQL clause and similarly, for an RXL constructor.

```
Constructor :- Const(String)
             | Var(String)
             | Element(Tag, [ Constructor ])
             | NestedQuery(XMLQL)
             | NestedQuery(RXL)
```

Finally, a canonical pattern is represented by the head variable (that occurs on the right-hand side of `in`), a regular-path expression over strings, and the target variable (that occurs in the body of an element):

```
Pattern :- Var, RegPathExpr, Var
```

In our example, each regular-path expression is one string atom, but in general, strings can be combined with the alternation (|), concatenation (.), and Kleene-star ($*$) operators, similar to those used in regular expressions. For example, the pattern `<company|organization>` matches a `company` or an `organization` element.

3.2 Partial Evaluation and Query Rewriting

The composition function `compose` in Figure 11 takes two *environments*, which are lists of variable, value pairs. The initial environment (`X_env`) maps the distinguished variable `$viewtree` to the root of the view tree referenced by the query. The initial map of XML-QL variables to RXL expressions (`XR_varmap`) is empty, and `X_clause` is the top-level XML-QL clause (lines 1–3). In our example, `$viewtree` is bound to the root of the tree in Figure 9. The result of `compose` is a list of RXL clauses. In the pseudo-code, XML-QL expressions are prefixed by `X_` and RXL expressions by `R_`.

Function `compose` (line 7) decomposes `X_clause` into its patterns, filters, and constructor, and rewrites each nested pattern in a canonical form as a list of unnested patterns. New variables are introduced to represent the intermediate nodes in the nested pattern. For example, the pattern in Figure 5 is rewritten as:

```
$viewtree "supplier" $t1,
$t1       "company"  $company,
$t1       "product"  $t2,
$t2       "name"     $name,
$t2       "retail"   $retail,
$t2       "sale"     $sale
```

where `$viewtree`, `$t1`, and `$t2` represent the root and two intermediate nodes.

On line 13, the patterns are evaluated in the current environment, producing `X_solution_relation`, which maps XML-QL variables to nodes in the view tree. Each tuple in `X_solution_relation` represents one possible rewriting of the XML-QL query over the view. For each `X_tuple`, the current environment is extended with the new variable bindings (line 19). In our example, there is only one pattern match and therefore one potential solution:

`$viewtree`	`$t1`	`$company`	`$t2`	`$name`	`$retail`	`$sale`
Root()	Supp()	Comp(”Acme...”)	Prod(cpid)	Name(cpid,*citem*)	Retail(cpid, *cprice*)	Sale(cpid, spid, *sprice*

Function `newVariables` (line 22) computes the new mappings of XML-QL and RXL variables to common RXL variables. We omit the details. In summary, `newVariables` recovers the correspondence between Skolem terms that share a common ancestor in the XML-QL pattern; this correspondences determines the mappings for RXL variables. For XML-QL variables, the mapping is simple: if the corresponding value is a leaf node, the variable is replaced by the underlined variable of the corresponding Skolem term; if the corresponding value is an internal node, the variable is replaced by the complete RXL expression that computes that element. In our example, `newVariables` produces the mapping:

```
$company -> "Acme Supplier", $retail -> cprice,
$name    -> citem,           $sale   -> sprice
```

All other variables remain unchanged. Lastly, function `oneSolution` (line 25) takes the new environment and computes the new RXL clause, which is appended to the list of other potential solutions.

Function `oneSolution` in Figure 12 constructs the RXL clause in three steps. First, for each XML-QL variable `X_v` in `X_clause`, it projects `X_v`'s value from the solution tuple `X_tuple`; its value is a view-tree element and an associated rule, whose head and body are projected in `R_head` and `R_body`. Function `makeCopy` (line 13) assigns fresh variable names to all free variables in `R_body`, i.e., those that do not occur in the rule's head. Function `rewriteR` (line 15) rewrites the new rule, using the variable name mapping `XR_varmap`. The new condition is added to the conjunctive conditions in `R_conditions`. Second, the function `rewriteX` (line 20) rewrites the XML-QL filters in `X_filters` and adds those to `R_conditions`. Finally, `rewriteX` (line 23) rewrites the XML-QL constructor to produce the new RXL constructor and the new RXL clause is returned.

In our example, there are four XML-QL variables `$company`, `$name`, `$retail`, and `$sale`, whose corresponding rules before variable renaming are:

```
// Top-level invocation of compose function
X_env = new [("$viewtree", Root())]
XR_varmap = new []
R_clause_list = compose(X_env , XR_varmap, X_clause)

fun compose(Env X_env, VarMap XR_varmap, XMLQL X_clause) : RXL list {
  (X_patterns, X_filters, X_constructor) = decompose(X_clause);

  // Get pairs of (head, target) variables from XML-QL patterns
  X_patvars = getHeadTargetMap(X_patterns);

  // Evaluate pattern on view tree
  X_solution_relation = evalPattern(X_patterns, X_env);

  // Consider each potential solution
  R_clauses = new []
  for each X_tuple in X_solution_relation {
    // Extend current environment with new variable bindings
    X_env' = appendList(X_env, X_tuple);

    // Compute new XR_varmap variable map from X_nodemap
    XR_varmap' = newVariables(X_env', X_patvars, XR_varmap);

    // Compute RXL clause for potential solution
    R_clauses = cons(oneSolution(X_env', XR_varmap', X_clause, X_tuple), R_clauses)
  }
  return R_clauses
}
```

Figure 11: Composition algorithm : top-level `compose` function

```
// Return new RXL clause for potential solution in X_tuple
fun oneSolution(Env X_env, VarMap XR_varmap, XMLQL X_clause, Env X_tuple) : RXL {
  R_conditions = new []
  // For each XMLQL variable X_v in X_clause
  foreach X_v in getVariables(X_clause) {
    // Get view-tree element bound to X_v
    R_elem = project(X_tuple, X_v);
    // Get body of rule associated with view-tree element
    (R_head, R_body) = R_elem;
    foreach R_rule in R_body {
      R_rule' = makeCopy(R_rule)
      // Rename head variables in R_rule' and add to R_conditions
      R_conditions = cons(rewriteR(XR_varmap, R_rule'), R_conditions)
    }
  }
  // Rename variables in X_filters and add to R_conditions
  foreach X_filter in X_filters
    R_conditions = cons(rewriteX(X_env, XR_varmap, X_filter), R_conditions)
  // Rename variables in X_constructor
  R_constructor = rewriteX(X_env, XR_varmap, X_constructor)
  // Construct new RXL clause: solution conditions + RXL constructor
  return new RXL(R_conditions, R_constructor)
}
```

Figure 12: Composition algorithm: `oneSolution` function

```
// rewriteX rewrites XML-QL expression as RXL expression
fun rewriteX(Env X_env, VarMap XR_varmap, X_Expr E) {
  fun substX(E) {
    case E of
      Var(v)                = lookupMap(XR_varmap, v)
      Const(c)              = new Const(c)
      Element(T,X)          = new Element(T, mapList(substX, X))
      Relop(op,E1,E2)       = new Relop(op, substX(E1), substX(E2))
      // Cases for all types of BoolExprs . . .
      // Recursively compose and rewrite nested XML-QL query
      NestedQuery(X_clause) = new NestedQuery(compose(X_env, XR_varmap, X_clause))
  }
  return substX(E)
}
// rewriteR renames RXL variables.
fun rewriteR(XR_VarMap varmap, R_Expr E) {
  fun substR(E) {
    case E of
      Var(v)                = lookupMap(XR_varmap, v)
      TableExpr(name, vars) = new TableExpr(name, mapList(substR, vars))
      Filter(b)             = new Filter(substR(b))
      Or(l1, l2)            = new Or(mapList(subst, l1), mapList(substR, l2))
      // Cases for all types of BoolExprs . . .
  }
  return substR(E)
}
```

Figure 13: Composition Algorithm: `rewrite` function

```
Supp()                    :- true
Name(cpid, citem)         :- Clothing(pid, citem, category, _, _), category = "outerwear"
Retail(cpid, cprice)      :- Clothing(cpid, _, category, _, cprice), category = "outerwear"
Sale(cpid, spid, sprice) :- Clothing(cpid, _, category, _, _), category = "outerwear",
                             SalePrice(spid, sprice), cpid = spid
```

There is one XML-QL filter:

```
$sale < 0.5 * $retail
```

The `rewriteX` and `rewriteR` functions in Figure 13 replace XML-QL and RXL variables by their new names in `XR_varmap`. The "helper" functions `substX` and `substR` perform the variable substitutions. Note that `rewriteX` calls `compose` recursively to rewrite a nested XML-QL clause into an equivalent nested RXL clause.

In our example, `R_conditions` after variable renaming is :

```
true,
Clothing(cpid, citem, x_category, _, _), x_category = "outerwear",
Clothing(cpid, _, y_category, _, cprice), y_category = "outerwear",
Clothing(cpid, _, z_category, _, _), z_category = "outerwear",
SalePrice(spid, sprice), cpid = spid
sprice < 0.5 * cprice
```

Although the composition algorithm produces redundant equations like these, we can use standard query minimization techniques and simplify this expression to:

```
  Clothing(cpid, citem, x_category, _, cprice), x_category = "outerwear",
  SalePrice(spid, sprice), cpid = spid
  sprice < 0.5 * cprice
```

Finally, `X_constructor`, bound to the construct clause in Figure 5, is rewritten, and the new RXL clause is:

```
where
  Clothing(cpid, citem, x_category, _, cprice),
  x_category = "outerwear",
  SalePrice(spid, sprice), cpid = spid
  sprice < 0.5 * cprice
construct
  <result ID=Result("Acme Supplier")>
     <supplier>"Acme Supplier"</supplier>
     <name>citem</name>
  </result>
```

This query is semantically equivalent to the composed view in Figure 6. The translator module translates this query into SQL and constructs the XML value as described in Section 2.4.

3.3 Aggregation Queries

We briefly describe how aggregations in XML-QL queries can be "pushed" into composed RXL views and evaluated by the target RDBMs. In both XML-QL and RXL, we use Skolem terms to specify the values by which aggregate expressions are grouped.

Suppose a reseller wants to count the total number of reports for each defective product. This can be expressed in XML-QL as follows:

```
where
  <supplier.product ID=$pid>
    <name>$n</>
    <report>$r</>
  </> in "http://acme.com/products.xml"
construct
  <product ID=F($pid)>
    <name ID=G($pid,$n)>$n</>
    <totaldefects ID=H($pid)>count(*)</>
  </>
```

The Skolem term `F($pid)` in `<product ID=F($pid)>` asserts that all bindings for the variables `$pid`, `$n` and `$r` are grouped by `$pid`'s value. Similarly, the Skolem term `H($pid)` specifies the grouping attributes for the aggregate function `count(*)`, which counts the total number of bindings. This idea is similar to the GROUP BY construct in SQL. XML-QL and RXL's semantics guarantee that only one element is produced for each value of a Skolem term, e.g., one `name` element is emitted for each value of `$n`.

We use a simple extension to datalog that accommodates aggregate functions [1]. An example of a datalog rule that uses a "generator" to count values is:

```
  C(p, q, COUNT(*)) :- R(p, q)
```

Only the last argument in the head can be an aggregate function; the other arguments specify the grouping attributes. The meaning is that C contains the set of triples (p, q, r) where r is the number of tuples in the group corresponding to values (p, q) in the relation R.

Using our composition algorithm, the XML-QL query above can be rewritten as:

```
from  Clothing $c, Problems $p
where $c.pid = $p.pid
construct
  <product ID=F($c.pid)>
    <name ID=G($c.pid, $c.item)>$c.item</>
    <totaldefects ID=H($c.pid)>count(*)</>
  </product>
```

Note that the aggregate function is "pushed" into the RXL view. When this view is materialized, the aggregation will be evaluated by the relational engine. Most importantly, this query can be evaluated efficiently, because commercial database systems are often highly optimized for aggregation queries.

4 Related Research

SilkRoute is an example of a mediator system, as defined by Geo Wiederhold [20]. Garlic [12] and Tsimmis [16] are well known examples of mediator systems for heterogeneous or semistructured data.

Bosworth [3] discusses the need for tools that export relational data into XML views. Several commercial tools already exist. The ODBC2XML tool allows users to define XML documents with embedded SQL statements, which allows them to construct an XML view of the relational data. Such views are materialized, however, and cannot be further queried with an XML query language like XML-QL. Alternatively, Oracle's XSQL tool defines a fixed, canonical mapping of the relational data into an XML document, by mapping each relation and attribute name to an XML tag and tuples as nested elements. Such a view could be kept virtual, but the approach is not general enough to support mapping into an arbitrary XML format.

XML-QL was first proposed as a W3C technical note [8], then published in [9]. It is the first complete query language specifically designed for XML, borrowing ideas from the research area on semistructured data [2, 4, 11]. Other query languages for XML include XSL [5, 6], XQL [19], Lorel [14], XMas [13], and YaTL [7].

Query composition is simple for select-project-join queries [18, 1], and for the relational calculus [1], in general. In the context of semistructured data, Papakonstantinou et al. first address the problem in the framework of MSL [15], a datalog-like language. Their composition algorithm, called *query decomposition and algebraic optimization*, uses a unification algorithm on the view's head and the query's body. Deutsch et al. [10] and Papakonstantinou and Vassalos [17] address query composition in the more complex setting of query rewriting for semistructured data. Our solution borrows ideas from [10].

5 Discussion

SilkRoute is a general, dynamic, and efficient framework for viewing and querying relational data in XML. We believe it is the first XML-export tool to support arbitrarily complex, virtual views of relational data and to support XML user queries over virtual views. The ability to support arbitrary views is critical in data exchange between inter-enterprise applications, which must abide by public DTDs and cannot reveal the underlying schemas of their proprietary data. The main contribution of this work is a sound and complete algorithm for composing virtual views and user queries. SilkRoute has many benefits: only the relational data requested by a user query is ever materialized; that data is always produced on demand; and the relational engine performs most of the computation efficiently.

We have not addressed two important open problems: general techniques for translating of RXL into efficient SQL and minimization of composed RXL views. As discussed in Sec. 2.4, SilkRoute has one translation strategy, which generates one SQL query for each RXL sub query, which must be in conjunctive-normal form (CNF). In practice, RXL view queries can be in disjunctive-normal form (DNF); for example, parallel RXL blocks often construct parts of complex elements independently, i.e., they express unions. User queries over such views often produce composed queries with many unions. Currently, we normalize any RXL sub-query in DNF into multiple sub-queries in CNF, which is a quadratic increase in the number of sub-queries to evaluate. In practice, we may be able to translate queries in DNF directly into SQL, for example, by using SQL's union-join constructs. Similarly, nested RXL queries often express left outer joins, e.g., the parent sub-query is the left relation and the child sub-query is the right relation. Currently, we generate two SQL queries, one for parent and child, but clearly one SQL query suffices. In addition to reducing the number of SQL queries, we want to minimize each individual RXL sub-query, i.e., eliminate all redundant expressions, so that the resulting SQL query is also minimal. Techniques exist for query minimization [1], but general algorithms are NP-complete. We expect heuristic algorithms to be effective for RXL queries, because RXL's nested block structure can help identify those expressions that most likely are redundant. Our future research will focus on these problems and on applying SilkRoute to large-scale applications.

Availability. XML-QL and SilkRoute are implemented in Java. SilkRoute has drivers for Oracle and MySQL database servers. XML-QL can be downloaded from `http://www.research.att.com/sw/tools/xmlql` and its users' guide is at `http://www.research.att.com/~mff/xmlql/doc`. SilkRoute should be available publicly in 2000.

Vitae. Mary Fernandez is Senior Technical Staff Member in Large-Scale Programming Research at AT&T Labs. Dan Suciu is Principal Technical Staff Member in Information Systems and Analysis Research at AT&T Labs. Wang-Chiew Tan is a Ph.D. candidate in Computer Science at University of Pennsylvania.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.

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

[3] A. Bosworth and A. L. Brown. Microsoft's vision for XML. *IEEE Data Engineering Bulletin*, pages 37–45, Sept 1999.

[4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[5] J. Clark. XML path language (XPath), 1999. `http://www.w3.org/TR/xpath`.

[6] J. Clark. XSL transformations (XSLT) specification, 1999. `http://www.w3.org/TR/WD-xslt`.

[7] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion ! In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 177–188, 1998.

[8] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for XML, 1998. `http://www.w3.org/TR/NOTE-xml-ql/`.

[9] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, Toronto, 1999.

[10] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[11] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[12] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of VLDB*, Athens, Greece, 1997.

[13] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definition and DTD inference for xml. In *Workshop on Semistructured Data and Nonstandard Data Formats*, January 1999.

[14] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, Edinburgh, UK, September 1999.

[15] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of Very Large Data Bases*, pages 413–424, September 1996.

[16] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[17] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 455–466, Philadephia, PA, June 1999. ACM Press.

[18] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw Hill, 2nd edition, 1999.

[19] J. Robie. The design of XQL, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

[20] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.