SilkRoute : A Framework for Publishing Relational Data in XML

MARY FERNÁNDEZ
AT&T Labs - Research
and
YANA KADIYSKA and DAN SUCIU
University of Washington
and
ATSUYUKI MORISHIMA
Shibaura Institute of Technology
and
WANG-CHIEW TAN
University of California at Santa Cruz

XML is the "lingua franca" for data exchange between inter-enterprise applications. In this work, we describe SilkRoute, a framework for publishing relational data in XML. In SilkRoute, relational data is published in three steps. First, the relational tables are presented to the database administrator in a canonical XML view. Second, the database administrator defines in the XQuery query language a public, virtual XML view over the canonical XML view. Third, an application formulates an XQuery query over the public view. SilkRoute composes the application query with the public-view query, translates the result into SQL, executes this on the relational engine, and assembles the resulting tuple streams into an XML document.

This work makes two key contributions to XML query processing. First, it describes an algorithm that translates an XQuery expression into SQL. The translation depends on a query representation that separates the *structure* of the output XML document from the *computation* that produces the document's content. The second contribution addresses the optimization problem of how to decompose an XML view over a relational database into an optimal set of SQL queries. We define formally the optimization problem, describe the search space, and propose a greedy, cost-based optimization algorithm, which obtains its cost estimates from the relational engine. Experiments confirm that the algorithm produces queries that are nearly optimal.

Authors' addresses: M. Fernández, 180 Park Ave., Florham Park, NJ 07932-0971, email: mff@research.att.com; Y. Kadiyska and D. Suciu, Computer Science Department, Univ. of Washington, Box 352350 Seattle, WA 98195, email: {yana,suciu}@cs.uwashington.edu; A. Morishima, Dept. of Info. Sci. and Eng., Shibaura Institute of Technology, 307 Fukasaku, Saitama-city, Saitama 330-8570, Japan, email: amori@sic.shibaura-it.ac.jp; W. Tan, Computer Science Department University of California at Santa Cruz Santa Cruz, CA 95064 email: wctan@saul.cis.upenn.edu.
Dan Suciu was partially supported by the NSF CAREER Grant 0092955, a gift from Microsoft, and an Alfred P. Sloan Research Fellowship. Wang-Chiew Tan contributed to this work while a visitor at AT&T Labs - Research and while a Ph.D. candidate at the University of Pennsylvania.

Categories and Subject Descriptors: H.2.3 [**Languages**]: Query languages; H.2.4 [**Systems**]: Query processing; D.3.2 [**Language Classifications**]: Very high-level languages

General Terms: Languages, Experimentation, Standardization

Additional Key Words and Phrases: XML, XQuery, XML storage systems

1. INTRODUCTION

XML is a Jack of many trades: It is a document mark-up language, an object-serialization format, a network message format, and most importantly, a universal 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 automotive, health care, and telecommunications, publish document type definitions (DTDs) and XML Schemata [World-Wide Web Consortium 2001a], which specify the format of the XML data to be exchanged between their applications. The aim is to use XML as a "lingua franca" for 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. Most existing data, however, is stored in non-XML database systems, so applications typically convert data into XML for exchange purposes. When received by a target application, XML data can be re-mapped into the application's data structures or target database system. Thus, XML often serves as a language for defining a *view* of non-XML data.

We are interested in the case when the source data is relational, and the exchange of XML data is between separate organizations or businesses on the Web. This scenario is common, because an important use of XML is in business-to-business applications, and most business-critical data is stored in relational databases. This scenario is also challenging, because it demands that frameworks for publishing XML meet three requirements: they must be general, selective, and efficient.

First, a publishing framework must be able to specify *general* mappings from relational data to XML. Relational data is flat, normalized (1NF and sometimes 3NF), and its schema is often proprietary. For example, relation and attribute names may refer to a company's internal organization, and this information should not be exposed in the exported XML data. In contrast, XML data is nested, unnormalized, and its DTD or XML Schema is public. Thus, the mapping from the relational model to XML is inherently complex. Some commercial systems fail to be general, because they map each relational database schema into a fixed, canonical XML schema. This approach is limited, because no public XML schema 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 XML schema.

A second requirement is that a publishing framework must be *selective*, 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*. An 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 database. Some com-

mercial products allow users to export relational data into XML by writing scripts. According to our definition, these tools are general but not selective, because the entire document is generated all at once.

A third requirement is that a publishing framework must be *efficient*. Relational query engines have sophisticated query optimizers and evaluation engines. A publishing framework must exploit the relational engine whenever data items in the XML view are materialized in XML documents.

In this work, we describe SilkRoute, a general, selective, and efficient framework for publishing relational data in XML. In SilkRoute, relational data is published in XML in three steps. First, the relational tables are presented to the database administrator in a canonical XML view. This step requires only the relational schema as input and is fully automated. In the second step, the database administrator specifies the *public XML view* of the relational database in XQuery [World-Wide Web Consortium 2002e]. XQuery is a powerful language and this allows SilkRoute to express XML views with a complex structure and with arbitrary levels of nesting. Since XQuery is defined on XML data, not on relational data, the input to the public query is a canonical XML view of the relational database. In the third step, an application formulates an XQuery query over the public view, extracting the XML data of interest to the application and structuring it in the format required by the application. The system converts that query into one or more SQL queries, executes them on the relational engine, and assembles their results into an XML document. SilkRoute's implementation is based on Galax [Choi et al. 2002], which is an XQuery implementation based on the XQuery Formal Semantics.

To implement the SilkRoute framework, this work makes two key technical contributions, which are discussed next.

XQuery to SQL Translation. We describe an algorithm that translates any XQuery expression into an equivalent set of one or more SQL queries. The set may contain more than one SQL query, because the XML answer is nested: each nesting level may be expressed by a different SQL query.

While XQuery was designed with database applications in mind, translating the full language into SQL is non-obvious. The main difficulty in the translation is that XQuery is compositional: a subquery can construct an intermediate XML result that is input to another subquery. This feature is evident even in simple XQuery queries and common in SilkRoute, in which the public query is composed with the application query. XQuery composition is hard to translate into SQL because there is no direct representation of the intermediate XML value in SQL.

Our solution to the translation problem is to represent XQuery expressions by an abstraction called a *view forest*. Semantically, a view forest defines a mapping from a relational database to an XML document. The key idea is to separate the *structure* of the output XML document from the *computation* that produces the document's content. The structure is represented by a forest whose nodes are labeled with XML element names, attribute names, or atomic types. The computation is represented by SQL queries that are attached to the nodes. No intermediate XML values are constructed by the view forest; only the final XML result is constructed. In addition, no data manipulation is performed in SilkRoute itself; the entire computation is pushed into the SQL queries.

We present a *view-forest composition* algorithm that translates any XQuery expression into a view forest. In short, the algorithm is a translation from XQuery to SQL, and therefore is of general interest beyond the scope of SilkRoute. The algorithm applies to a large, useful subset of the XQuery language. It excludes recursive functions, features that depend on the document order of values in XML documents (e.g., the "before" and "after" operators (<</>>) and the `is/isnot` operators), and features that enforce an order based on data values (e.g., the `sort` expression). An XQuery expression typically specifies the relative order of elements in the output document and this order is preserved.

The view forest is used at several levels in SilkRoute. The canonical XML view of the input relational database, the public query defining the public XML view, and the application query applied to the public XML view are all represented as view forests. The fact that the same abstraction serves *all* these roles is an elegant property of the view forest.

The Optimization Problem for XML Publishing. When relational data is mapped into XML, there are several ways in which one can compute the XML view as a combination of SQL queries. Each set of SQL queries that can compute the XML view is called a *plan*. Selecting an efficient plan is an optimization problem that is similar in spirit, although different in details, to the traditional query-plan optimization problem. We address the optimization problem for XML publishing, describe the search space, and propose a heuristic based optimization algorithm.

A view forest defines a mapping from relational data to XML. Any partition of the forest into connected components corresponds to an execution plan: for each component we derive one SQL query by essentially outer-joining and outer-unioning the SQL queries on the nodes in that component. Thus, the search space of the optimization problem consists of all partitions of the view forest. We call each such partition a *view-forest decomposition*. There are trade-offs in choosing a good decomposition. The coarsest decomposition (only one partition) results in a single SQL query, which needs only one connection to the database system, but which requires the relational engine to optimize and execute a a very complex SQL query. The finest decomposition (each node is a separate partition) results in several multiple select-project-join queries, but requires many connections to the database system and may produce redundant computations in the set of SQL queries.

In general, there are an exponential number of possible plans for decomposing a view forest into one or more SQL queries. We propose a greedy algorithm for selecting a good decomposition plan; the algorithm takes as input estimates of query and data cost produced by the relational query engine. We evaluated our algorithm experimentally and concluded that the plan-selection algorithm produces queries that are nearly optimal.

In the next section, we present an example scenario from electronic commerce. In Section 2, we describe SilkRoute's architecture, its various components, and the subset of XQuery supported by SilkRoute. Section 3 defines the view forest abstraction and Section 4 gives the complete view-forest composition algorithm. In Section 5, we present our greedy algorithm for query decomposition and give experimental results that support the efficacy of the algorithm. Finally, Section 6 describes alternative techniques and systems and discusses the impact of SilkRoute.

```
element supplier {
  element company,
  element product*
}

element product {
  element name,
  element category,
  element description,
  element retail,
  element sale?,
  element report*
}

element company { string }
element name    { string }
element category  { string }
element description { string }
element retail  { float }
element sale    { float }
element report  {
  attribute code { string },
  string
}
```

Fig. 1. Schema of XML data exported by suppliers to resellers in XQuery's type syntax.

```
CREATE TABLE Clothing(
  pid CHAR(10) PRIMARY KEY,
  item VARCHAR(30),
  category VARCHAR(20),
  description VARCHAR(200),
  price REAL,
  cost REAL)

CREATE TABLE Discount(
  pid CHAR(10) PRIMARY KEY,
  item VARCHAR(30),
  discount REAL)

CREATE TABLE Problems(
  pid CHAR(10),
  code CHAR(10),
  comments VARCHAR(200))
```

Fig. 2. Schema of supplier's relational database.

1.1 An Example

In our example scenario from electronic commerce, product *suppliers* provide information to product *resellers*. For their mutual benefit, suppliers and resellers have agreed to exchange data in a format that conforms to a particular XML schema. The shared schema is depicted Figure 1 in the type notation of XQuery [World-Wide Web Consortium 2002c]. Every `supplier` element contains a `company` element followed by a possibly empty sequence of `product` elements. The sequence operator (`,`) combines elements into a sequence; the repetition operator (`*`) denotes zero or more instances of an element. A `product` element, in turn, contains several other elements and an optional `sale` element. The optionality operator (`?`) indicates that there can be zero or one `sale` element in `product`. The `company`, `name`, `category`, and `description` elements all contain a single string. A trouble report includes a `code` attribute, containing a string that indicates the class of problem; the report's content is a string, which is the customer's comments. Most importantly, this schema is used by suppliers and resellers, and it is a public document. Typically, a public schema is written in XML Schema notation [World-Wide Web Consortium 2001a; 2001b], which is translated by an XQuery processor into its internal type system. Because XML Schema syntax is verbose, we describe our example schema in the more concise notation of XQuery's internal types. We assume, as usual, that the XML schema was designed by agreement among many (possibly hundreds) suppliers and resellers, and does not reflect in any way how the data is organized by particular suppliers or resellers.

Consider now a particular supplier whose business data is organized according to the relational schema depicted in Figure 2. There are three tables: `Clothing` contains tuples corresponding to products; the `Discount` table contains product discounts; and the `Problems` table contains trouble codes of products and their

reports. This is a first-normal form relational schema, designed for the supplier's particular business needs. We stress that the XML schema and the relational schema were designed by different organizations, and with different purposes. As a result, the supplier needs to convert data from its relational representation into the XML representation, and make the XML view available to resellers. In our example, we assume the supplier exports only a subset of its inventory, in particular, its stock of outer-wear that it wants to sell at a reduced price at the end of the winter season.

Resellers need not be aware of how the supplier provides the XML data conforming to the schema in Figure 1. They formulate their queries over the XML view assuming that it conforms to the agreed schema. Some examples of queries are:

—Retrieve products whose sale price is less than 50% of retail price.

—Count the number of "defective" reports for a product.

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

As these queries suggest, the reseller is typically interested in a subset of the data 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 across suppliers and are not accessible by the reseller.

2. ARCHITECTURE

The architecture of SilkRoute is depicted in Figure 3. It serves as middle-ware between a relational database and an application accessing that data over the Web. First, the relational tables are presented to the database administrator in a virtual, *canonical XML view*. The database administrator then writes a *public query* over the canonical XML that defines the *public XML view*. The public query is typically complex, because it transforms the relational data into deeply nested XML data. Applications only have access to the public XML view, not the relational database.

To access the data, an application programmer formulates an *application query* over the XML view; this query is composed with the public query, resulting in a new query. This composed query is submitted to the query planner, which translates it into one or more SQL queries that are executed by the relational engine, producing one or more tuple streams. The XML generator merges these tuple streams into one virtual relation and constructs an XML document by nesting and tagging tuples in the virtual relation. The final XML document is returned to the application. The only data manipulation performed by SilkRoute is merging of sorted tuple streams. Any complex data processing is pushed into the relational engine.

Both the public query and the application queries are expressed in XQuery. The entire XQuery language is supported, except for recursive functions and features that depend on the XML document order. The precise XQuery fragment supported by SilkRoute is described in Section 4.

Internally, XQuery expressions are represented by *view forests*. The relational database instance is represented as a *canonical view forest*; the public view is represented as the *public view forest*; and, finally, the application query is composed with the public view forest and results in an *application view forest*. The view forest is defined formally in Section 3. Except for the canonical view forest, which is generated directly from the relational schema, all other view forests are obtained

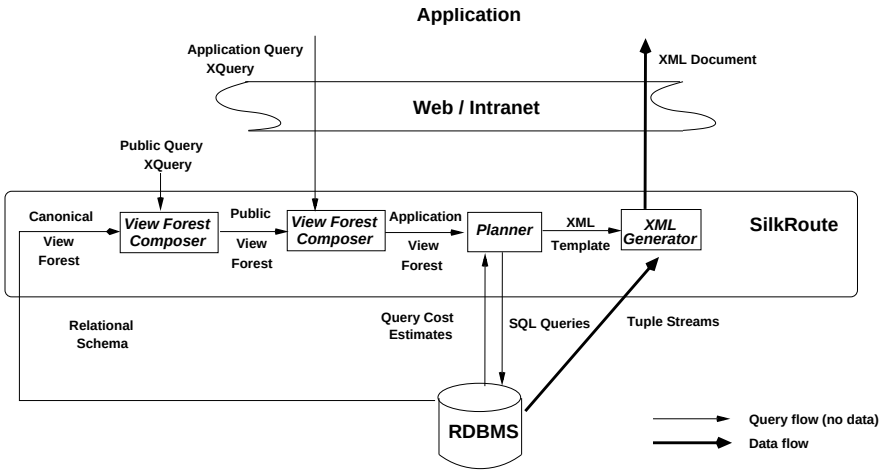


Fig. 3. SilkRoute's Architecture.

by composing an existing view forest with an XQuery expression. The *view forest composer* module is described in Section 4.

The scenario shown in Figure 3 is probably the most common use of SilkRoute, but minor changes to this information flow permit other scenarios. For example, the data administrator may export the entire database as one, large XML document by materializing the public query. This can be done by passing the public view forest directly to the planner. Unlike the application query, the public query may export a large part or possibly all of the relational database, therefore it is imperative that the planner choose a set of SQL queries that can be evaluated efficiently by the relational engine. In another scenario, the application view forest could be kept virtual for later composition with other application queries. This is useful when one wants to define a new XML view from an existing application view.

In the rest of this section, we describe SilkRoute's components, illustrating their functionality on our example in Section 1.1.

2.1 Canonical XML View of Relational Data

The relational database is represented in SilkRoute as a virtual, canonical XML view: this permits the database administrator to write XQuery expressions over the relational data. The view is virtual, because it is never materialized, and it is canonical, because the same rules are applied to convert any relational table to an XML view.

A relation with schema `R(A1,...,An)` and containing the tuples `(a11,...,a1n)`, ..., `(ak1,...,akn)` are mapped into a canonical XML view with the following form:

```
<R>
   <Tuple><A1>a11</A1>...<An>a1n</An></Tuple>
   ...
   <Tuple><A1>ak1</A1>...<An>akn</An></Tuple>
</R>
```

```
element CanonicalView {
  element Clothing,
  element Discount,
  element Problems
}

element Clothing {
  element Tuple {
    element pid { integer },
    element item { string },
    element category { string },
    element description { string },
    element price { float },
    element cost { float }
  }*
}

element Discount {
  element Tuple {
    element pid { integer },
    element item { string },
    element discount { float }
  }*
}

element Problems {
  element Tuple {
    element pid { integer },
    element code { string },
    element comments { comments }
  }*
}
```

Fig. 4. Canonical XML view of relational schema in Figure 2 using XQuery type notation.

SilkRoute uses the relational schema to generate the canonical XML view, as shown in Figure 3. For our example, Figure 4 shows the XML types for the canonical XML view of the relations given in Figure 2.

2.2 The Public XML View and the Public Query

The database administrator defines the public XML view by writing an XQuery expression over the canonical XML view. XQuery is a functional query language for XML that is statically typed and compositional. It incorporates all of XPath 2.0 [World-Wide Web Consortium 2002d] as a proper sub-language. XQuery is currently a W3C working draft [World-Wide Web Consortium 2002b]. The subset of XQuery supported by SilkRoute is defined formally in Section 4.

The SilkRoute-specific variable `$CanonicalView` is used to access the canonical XML view. We illustrate several XQuery features for defining public views, then describe the public query for the example in Figure 6.

Flat XML Views In the simplest case, the XML view is flat, like relational data. The public query consists of a `for-let-where-return` (`flwr`) expression, which iterates over tuples in the canonical XML view, applies predicates to the tuples, and constructs new XML values that occur in the public view. For example, the following query defines a fragment of the supplier's public view in Figure 2:

```
for $c in $CanonicalView/Clothing/Tuple
where data($c/category) = "outerwear"
return <product>
         <name>{ data($c/item) }</name>
         <category>{ data($c/category) }</category>
         <retail>{ data($c/price) }</retail>
       </product>
```

The `$CanonicalView` variable is bound to the canonical XML view conforming to

the schema given in Figure 4. The expression `$CanonicalView/Clothing/Tuple` denotes the set of `Tuple` elements in the `Clothing` relation. The `for` expression binds the variable `$c` to each tuple in this collection and for each binding, the `where` expression is evaluated. For every true evaluation of `where` expression, the `return` expression is evaluated, and all the `return` values are concatenated into a sequence of XML elements. The query produces an XML fragment like the following:

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

in which there is one `product` element for each tuple that satisfies the predicate in the `where` expression.

Nested XML Views When the XML view is nested, then the corresponding XQuery has nested `FLWR` expressions. An example is:

```
<view> {
  for $c in $CanonicalView/Clothing/Tuple
  return <product>
           <name>{ data($c/item) }</name>
           { for $p in $CanonicalView/Problems/Tuple
             where $p/pid = $c/cid
             return <report>{ $p/comments }</report> }
         </product>
} </view>
```

The outer-most constructor expression constructs the root element `view`. The first sub-expression constructs one `product` element for each tuple in `Clothing`. Its inner sub-expression creates zero or more `<report>` sub-elements, one for each report associated with that product. Readers familiar with SQL may recognize this expression as a left-outer join of `Clothing` with `Problems` followed by a group-by on `Clothing`.

Complex XML Views In some applications, the XML view requires complex restructuring of the relational data, e.g., when data from several relations needs to be fused. Assume that we want to compute a view containing all products in `Clothing` that are not discounted, all products in `Clothing` that are discounted (and include the discount) and all products that are in `Discount`, but do not occur in `Clothing`. Readers familiar with SQL will recognize this expression as a full outer join on `Clothing` and `Discount`. This view can be expressed in XQuery with parallel `let` expressions, `UNION`, and the `distinct-value` function. An example is shown in Figure 5. Here, the product ids from the `Clothing` relation are merged with the product ids from `Discount`, and the XML view contains one `<fused-product>` element for each distinct product id. Notice how this is achieved with intermediate XML values, `UNION`, and `distinct-value`.

The first two `let` expressions construct two collections of XML values independently and these collections are subsequently "merged" into one collection. The first `let` expression creates elements of the form:

```
<product id="i"><price>p</price></product>
```

```
<view> {
  let $items     := for $c in $CanonicalView/Clothing/Tuple
                    return  <product id={ data($c/pid) }>
                              { $c/price }
                            </product>,
      $discounted-items := for $d in $CanonicalView/Discount/Tuple
                    return  <product id={ data($d/pid) }>
                              { $d/discount }
                            <product>,
      $allitems := $items UNION $discounted-items,
      $prodids := distinct-value($allitems/product/@id)
  for $pid in $prodids
  return
    <fused-product id="{ $pid }">
      { for $p in $allitems/product
        where $pid = data($p/@id)
        return $p/*
      }
    </fused-product>
} </view>
```

Fig. 5. A public query performing element fusion

for each product id i in `Clothing`. The second `let` expression creates elements of the form:

```
<product id="i"><discount>d</discount></product>
```

for each product id i in `Discount`. The third `let` expression binds `$allitems` to the union of these two collections. The fourth `let` expression computes the set of distinct product ids occurring in `$allitems`. For each such id, a `fused-product` element is created, which contains the children elements of each product (`$p/*`) in `$allitems` with the same id. When the same product occurs both in `Clothing` and `Discount`, then the two corresponding `product` elements are merged into:

```
<fused-product id="i">
  <price>p</price><discount>d</discount>
</fused-product>
```

Public XML View in the Running Example. Figure 6 contains the public query for the example first described in Section 1.1. Lines 1, 2, and 26 create the root `<supplier>` element. The constructor expression on line 2 creates its `company` child element. The first nested expression (lines 3–25) contains the query fragment described above, which constructs one `product` element for each "outerwear" item. Within this expression, the nested expression (lines 12–16) expresses an outer join between the `Clothing` and `Discount` tables and constructs a `sale` element with the product's sale price nested within the outer `product` element. The last nested expression (lines 17–23) expresses an outer join between the `Clothing` and `Problem` tables and constructs one `report` element containing the problem code and customer's comments; the `report` elements are nested in the outer `product` element.

```
 1. <supplier>
 2.   <company>Acme Clothing</company>
 3.   {
 4.     for $c in $CanonicalView/Clothing/Tuple
 5.     where data($c/category) = "outerwear"
 6.     return
 7.       <product>
 8.         <name>{ data($c/item) }</name>
 9.         <category>{ data($c/category) }</category>
10.         <description>{ data($c/description) } </description>
11.         <retail>{ data($c/price) }</retail>
12.         { for $d in $CanonicalView/Discount/Tuple
13.           where $d/pid = $c/pid
14.           return
15.             <sale>{ data($c/price) * data($d/discount) }</sale>
16.         }
17.         { for $p in $CanonicalView/Problems/Tuple
18.           where $p/pid = $c/pid
19.           return
20.             <report code="{ data($p/code) }">
21.               { $p/comments }
22.             </report>
23.         }
24.         </product>
25.   }
26. </supplier>
```

Fig. 6. Public query (Q_P). The highlighted fragment forms another public query used in Fig. 17.

2.3 Application Query

Applications access the public XML data by formulating a query over the public view; the relational data cannot be accessed directly. Like the public query, the application query is written in XQuery, but the former is applied to the canonical XML view, while the latter is applied to the public XML view.

We illustrate with an application query in which the reseller retrieves all products with sale price less than half of retail price. The application query is shown in Figure 7. The variable `$PublicView` denotes the public view. One `supplier` element is created for each `supplier` element in the exported view. It contains one `name` and `discounted` element, which in turn contains one `product` for each discounted product. Note that the answer to the application query includes a small fraction of the relational database, i.e., only those products that are heavily discounted.

A note on static typing. One of XQuery's features is that the language is statically typed. SilkRoute uses typechecking both in the public query and in the application query. The public query is usually large, and increases with the size of the XML schema or DTD. In practice, XML schemas often have hundreds of elements or more [Sahuguet 2000], and specifying an XML public view of such schemas might

```
 1.  for $s in $PublicView/supplier,
 2.  return
 3.    <supplier> {
 4.      <name>{ data($PublicView/supplier/company) }</name>
 5.      <discounted>
 6.      { for $p in $s/product
 7.        where data($p/sale) < 0.5 * data($p/retail)
 8.        return <product>{ data($p/name) }</product>
 9.      }
10.      </discounted>
11.    } </supplier>
```

Fig. 7. Application query (Q_A).

require hundreds to thousands of lines of XQuery code. The static typechecker is essential for catching errors in such a query. In our example, the typechecker verifies that the public query in Figure 6 conforms to the schema in Figure 1. For example, had we omitted line 11, the typechecker would report that `product` must contain a `retail` subelement. Similarly, a typechecker can catch errors in an application query. For example, if we mistype line 7 of the application query as:

```
7.        where data($p/price) < 0.5 * data($p/retail)
```

then, based on the XML schema in Figure 1, the typechecker would report that `price` is an invalid subelement of `product`.

2.4 View-Forest Composer

The application query is composed with the public query before it is evaluated on the relational data. Because XQuery is compositional and both queries are XQuery expressions, they can be composed trivially by simply substituting the variable `$PublicView` by its definition in XQuery. The composer module, however, does more than syntactic substitution: It removes unnecessary expressions from the public view, eliminates any intermediate XML results, and computes the appropriate SQL queries for the relational database. Notice that the composer is used twice in Figure 3. In general, it can be applied whenever a new XML view is defined in terms of an existing one.

Before defining view forests and view composition formally, we illustrate composition informally using XQuery expressions. The composition module takes the public query in Figure 6 and the application query in Figure 7 and constructs the query is shown in Figure 8. We stress that the composition algorithm manipulates view forests, not XQuery expressions, and this example only illustrates the intuition. The composed query illustrates several important points. First, it combines fragments of the public query and application query, with the latter highlighted in Figure 8. Second, the composed query takes as input (the canonical XML view of) the relational database, and does not refer to the public view. The application query's predicate: `data($p/sale) < 0.5 * data($p/retail)`, is re-expressed in terms of the canonical relational view as:

```
data($c/price) * data($d/discount) < 0.5 * data($c/price).
```

Finally, the composed query structures the result as in the application query.

```
<supplier>
  <name> Acme Clothing </name>
  <discounted>
    { for $c in $CanonicalView/Clothing/Tuple,
      $d in $CanonicalView/Discount/Tuple
      where data($c/category) = "outerwear",
            data($c/pid) = data($d/pid),
            data($c/price) * data($d/discount) < 0.5 * data($c/price)
      return
        <product> { data($c/item) } </product>
    }
  </discounted>
</supplier>
```

Fig. 8. Composed query (Q_C), obtained by composing Q_P and Q_A. The highlighted fragment is obtained from Q_A, the rest from Q_P.

2.5 Planner

The planner takes a view forest and decomposes it into one or more SQL queries and an XML template. For example, the composed query in Figure 8 is translated into the SQL query:

```
SELECT  c.pid as pid, c.item as item
FROM    Clothing c, Discount d
WHERE   c.category = "outerwear",
        c.pid = d.pid,
        c.price * d.discount < 0.5 * c.price
ORDER BY c.pid
```

and into the XML template:

```
<supplier>
   <name>Acme Clothing</name>
   <discounted>
      <product> { $item } </product>
   </discounted>
</supplier>
```

where `$item` refers to the attribute `item` in the SQL query's `SELECT` clause; we describe the meaning of an XML template in more detail in Section 3. The SQL queries are executed by the relational engine, and the results are merged and converted into XML by the *XML generator*.

In this example, only one SQL query is required. In general, there are many ways to decompose a complex view forest into one or more SQL queries. Each SQL query has a `sort by` clause, making it possible for the XML generator to merge the tuple streams into an XML document in a single pass. Choosing an efficient evaluation strategy is important when the view query returns a large result. When searching for an efficient evaluation plan, the planner consults the relational query engine, which returns estimates of query cost.

3. VIEW FOREST

We introduce here the *view forest*, our representation of XQuery to SQL translation, which is used throughout SilkRoute. Semantically, a view forest defines a mapping from a relational database to an XML document. It separates the *structure* of the XML document from the *computations* that produce the atomic values contained in the document. The computations are expressed in SQL.

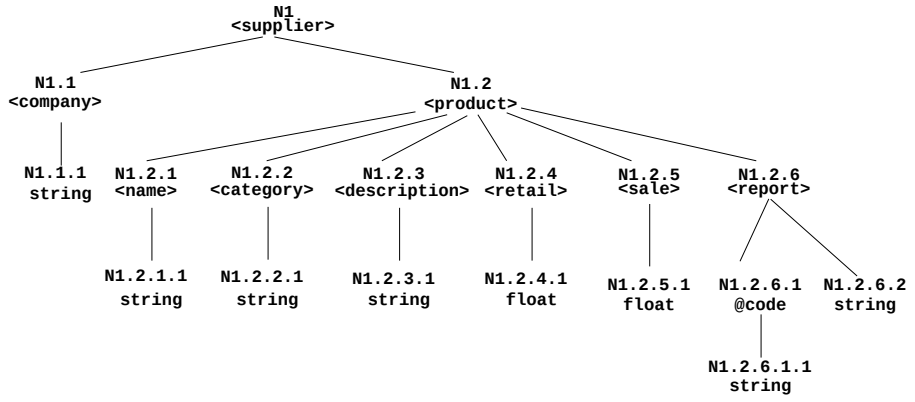
Any XQuery expression over a canonical view of a relational database can be rewritten as a view forest, and in SilkRoute, both the public query and the application query, which is composed with the public query, are represented as view forests. For example, the public query in Figure 6 is represented internally by the view forest in Figure 9, which we explain below.

We now define formally the view forest. Given a relational schema S, a *view forest* V is a forest[1] in which each node is labeled with an XML label and a SQL query fragment over S. The XML label and the SQL fragment differ slightly for element or attribute (internal) nodes and for atomic value (leaf) nodes. For an internal node, the XML label is an element or attribute name, and the SQL fragment consists of a required FROM clause and an optional WHERE clause. For a leaf node, the XML label is an atomic type (as specified in XML Schema [World-Wide Web Consortium 2001b]) and the SQL fragment consists of a required SELECT clause, containing a single value of the corresponding atomic type, and optional FROM and WHERE clauses. For exposition purposes, we use only the `string` atomic type and assume all other values are cast into strings. We require that a tuple variable is bound to each table occurring in a FROM clause, and that every tuple variable used in the SQL fragment of some node n is bound in the FROM clause of n or in the FROM clause of one of n's ancestors. The nodes, edges, and XML labels of the forest represent the *structure* of the XML view, while the SQL query fragments represent the *computation* performed by the relational database in order to construct the view. View forests exist because XQuery supports *sequences* whose items may be arbitrary XML elements. In practice, view forests are often trees, and then we refer to them as *view trees*.

We illustrate a view forest on our example public query Q_P in Figure 6; the corresponding view tree V_P is in Figure 9 (a). We associate to each node a unique identifier based on the Dewey encoding [Online Library]. This encoding allows us to check the parent-child relationship easily. For example, the nodes with identifiers $N1.2.1$ and $N1.2.2$ are siblings and are children of the node $N1.2$. The XML label for internal nodes is an element or an attribute name; examples are `<supplier>`, `<product>`, `@code`. The SQL fragments are in Figure 9 (b)[2]. The SQL fragments of the internal nodes $N1.2$, $N1.2.1$, and $N1.2.5$ contain FROM and WHERE clauses, whereas the leaf nodes $N1.2.1.1$ and $N1.2.6.1.1$. contain only SELECT clauses. The FROM or WHERE clauses may be empty, in which case we omit them (e.g., in the leaf nodes), or represent them with FROM () (e.g., $N1.2.1$). These queries are *fragments*: a WHERE or SELECT clause in a fragment may have tuple variables

[1]Recall that a forest is a graph in which for every two nodes x, y there exists at most one path from x to y. A forest with a single connected component is a tree.

[2]To distinguish between XQuery and SQL expressions, we use the `text` font to denote XQuery expressions and the SMALL-CAPS font to denote SQL expressions.



(a)

```
N1(<supplier>)         :- FROM ()
N1.1(<company>)        :- FROM ()
N1.1.1(string)         :- SELECT "Acme Clothing"
N1.2(<product>)        :- FROM Clothing c WHERE c.category ="outerwear"
N1.2.1(<name>)         :- FROM ()
N1.2.1.1(string)       :- SELECT c.item
N1.2.2(<category>)     :- FROM ()
N1.2.2.1(string)       :- SELECT c.category
N1.2.3(<description>) :- FROM ()
N1.2.3.1(string)       :- SELECT c.description
N1.2.4(<retail>)       :- FROM ()
N1.2.4.1(float)        :- SELECT c.price
N1.2.5(<sale>)         :- FROM Discount d WHERE d.pid = c.pid
N1.2.5.1(float)        :- SELECT d.discount * c.price
N1.2.6(<report>)       :- FROM Problems p WHERE p.pid = c.pid
N1.2.6.1(@code)        :- FROM ()
N1.2.6.1.1(string)     :- SELECT p.code
N1.2.6.2(string)       :- SELECT p.comments
```

(b)

Fig. 9. View forest V_P for public query Q_P in Figure 6: (a) tree representation (b) internal representation.

that are not defined in that fragment, however, each such tuple variable must be defined in the FROM clause of an ancestor. For example, node $N1.2$ defines the tuple variable `c`, which is used in in the WHERE clauses of $N1.2.2.1$ and $N1.2.6$.

To define formally the view-forest mapping from instances of S to XML, we need a notation. We associate with each node n a *complete* SQL query, C_n, as follows. The FROM clause of C_n is the concatenation of all FROM clauses of n and all n's ancestors; the WHERE clause of C_n is the conjunction of all WHERE clauses of n and all n's ancestors; and if n is a leaf, the SELECT clause of C_n is that of n, otherwise it is SELECT *. Notice that C_n is *complete*, i.e., all tuple variables used in C_n are defined in the FROM clause. Moreover, if n_1 is the parent of n_2, then all tuple variables bound in the FROM clause in C_{n_1} are also bound in the FROM

clause of C_{n_2}. Finally, notice that C_n is of the form SELECT-FROM-WHERE, not SELECT-DISTINCT-FROM-WHERE, thus duplicate values may occur in the answer.

Now we can define formally the mapping defined by a view forest. Given a database instance I of S, the XML tree $V(I)$ is defined as follows. $V(I)$'s nodes consist of the disjoint union of all answers $C_n(I)$, for all nodes n in the view forest. More precisely, $V(I)$'s nodes are pairs (n, a), where n is a node in V and a is a row in the answer of C_n on I: $a \in C_n(I)$. The parent-child relationship is defined as follows: (n_2, a_2) is the child of (n_1, a_1) if n_2 is the child of n_1 in V, and the answers a_1 and a_2 correspond to precisely the same values of the tuple variables defined in C_{n_1}. Finally, the XML label of a node (n, a) is the label of n in V, when n is an internal node, and is the string value a, if n is a leaf in V. The XML output tree is unordered. This completes the definition of $V(I)$. In general, $V(I)$ is an XML forest, but it is a tree when V is a tree and the SQL fragment of its root is empty.

Using the example in Figure 9, some instances of C_n are shown below:

$$
\begin{aligned}
C_{N1.2} = C_{N1.2.1} \;=\; & \texttt{SELECT *} \\
& \texttt{FROM Clothing c} \\
& \texttt{WHERE c.category = "outerwear"} \\
C_{N1.2.1.1} \;=\; & \texttt{SELECT c.item} \\
& \texttt{FROM Clothing c} \\
& \texttt{WHERE c.category = "outerwear"} \\
C_{N1.2.6} = C_{N1.2.6.1} \;=\; & \texttt{SELECT *} \\
& \texttt{FROM Clothing c, Problems p} \\
& \texttt{WHERE c.category = "outerwear" AND p.pid = c.pid} \\
C_{N1.2.6.1.1} \;=\; & \texttt{SELECT p.code} \\
& \texttt{FROM Clothing c, Problems p} \\
& \texttt{WHERE c.category = "outerwear" AND p.pid = c.pid}
\end{aligned}
$$

To compute the XML view, we execute all these SQL queries, and construct a distinct XML node for each row in each answer. For example, there will be one XML node for each row in $C_{N1.2.1}$, and its label is `<name>`, and there will be one XML node for each row in $C_{N1.2.1.1}$, labeled with the string `c.item`. To illustrate the parent-child relationship, consider the queries $C_{N1.2}$ and $C_{N1.2.6}$: a row in $C_{N1.2}$ is a parent of a row in $C_{N1.2.6}$ if and only if these two rows correspond to the same binding of the tuple variable `c`. In particular, a row in $C_{N1.2}$ may have multiple children, corresponding to the same binding of the tuple variable `c`, and to different bindings of `p`.

Next, we give examples of several view forests, showing their expressive power, usage in SilkRoute, and hint at the complexity of constructing them.

View forest for a canonical mapping. The simplest example of a view forest is the canonical mapping from a relational database to an XML view described in Section 2.1. Figure 10 gives a fragment of the view forest for the canonical XML view in Figure 2, corresponding to the `Clothing` relation. The fragments corresponding to `Discount` and `Problems` are similar and omitted. Such canonical view forests can be constructed automatically from the relational schema and are used

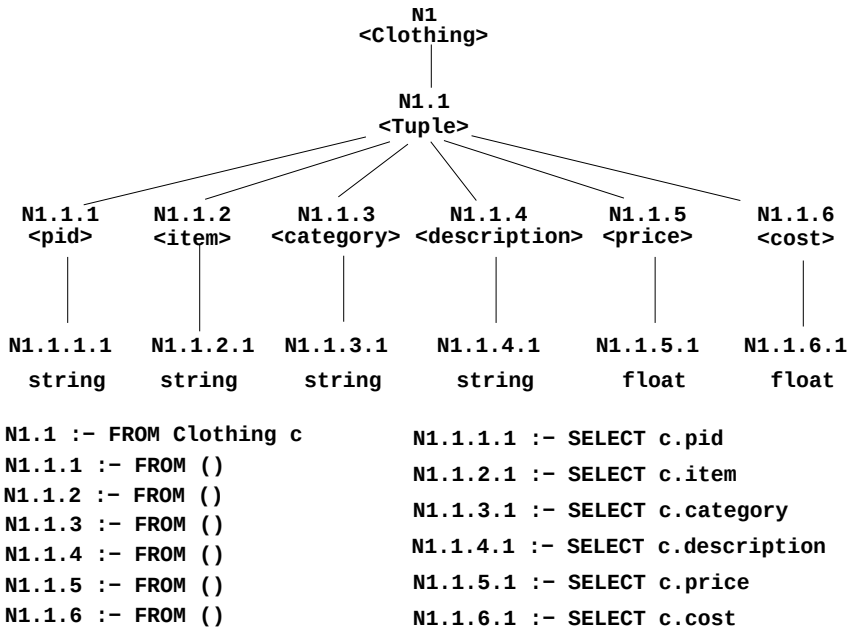


Fig. 10. Fragment of the canonical view forest for the `Clothing` relation in Figure 2.

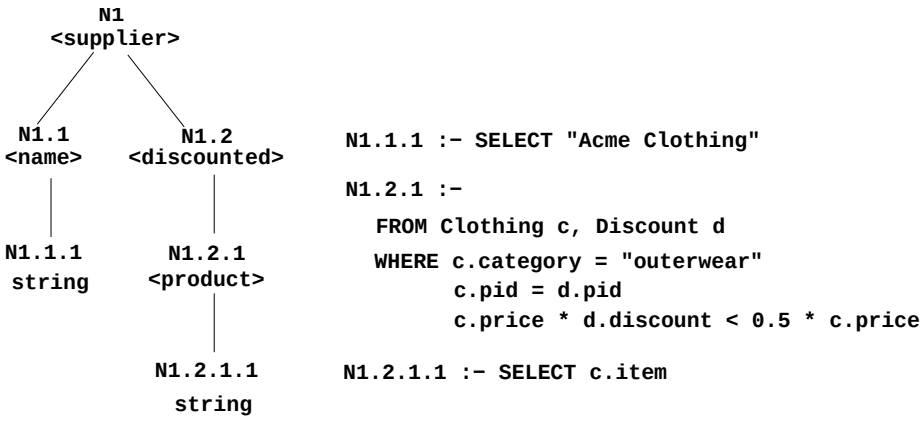


Fig. 11. View forest of composed query in Figure 8.

as starting points to construct more complex view forests, using the composition algorithm in Section 4.

View forest for the application query. Figure 11 gives the view forest for the composed query in Figure 8. Note that the SQL fragments for the `supplier`, `name`, and `discounted` elements are empty, because these nodes are always constructed. Also note that the child of the `name` node is the literal string "Acme Clothing". The SQL fragment for the `product` node expresses a join between the `Clothing`

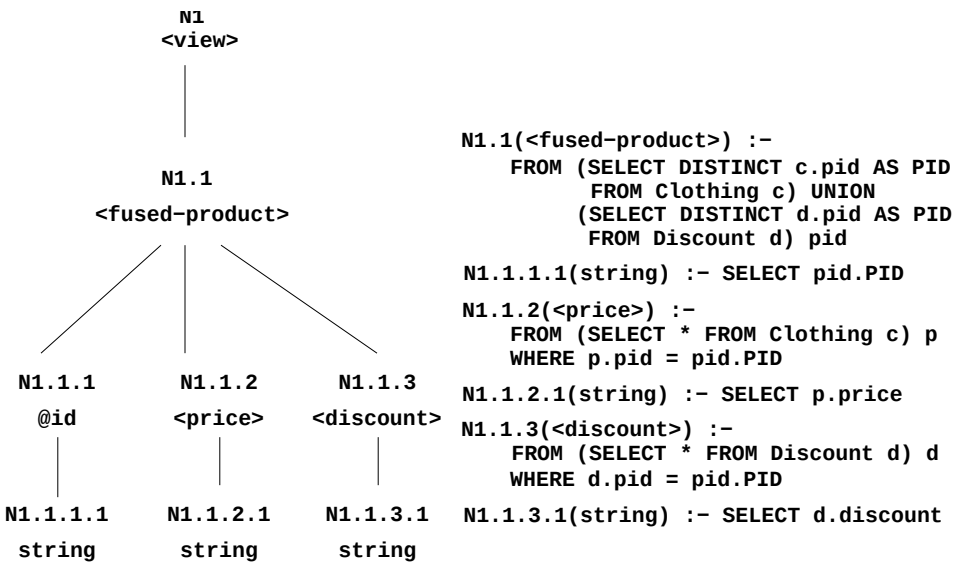


Fig. 12. View forest for the public query in Figure 5.

and `Discount` relations and selects each item in the outerwear category whose discounted price is less than half its retail price.

View forest for complex XQuery expressions. An important property of the view forests is that they can represent *any* XQuery expression in SilkRoute's subset. We illustrate with a more complex example, corresponding to the XQuery expression in Figure 5; the view forest is shown in Figure 12. The XQuery expression and the view forest differ significantly. In XQuery, an intermediate XML value is constructed, then the function `distinct-values` is applied, and the result is submitted to another XQuery subquery. In the view forest, there is no intermediate XML value; only the structure of the final XML answer is represented. The computations applied to the intermediate XML values are pushed into the SQL queries.

It is significant that *every* XQuery query can be transformed into a view forest representing the constructed XML document, with all the computations corresponding to intermediate results, nested queries, function applications, etc., pushed into the SQL queries. This property establishes a tight and precise connection between SQL and XQuery. As this example suggests, it is not obvious how to translate an arbitrary XQuery expression into a view forest. We show how to do so next.

4. VIEW-FOREST COMPOSITION ALGORITHM

Next, we present the view-forest composition algorithm (VFCA), which translates an XQuery query into a view forest. The algorithm expects the query to be expressed in a subset of XQuery, called XQueryCore, which is defined below. Because XQueryCore is compositional, the algorithm can be used to perform composition of canonical, public, and application queries.

We shall formalize VFCA next. Recall that a view forest V defines a mapping

from a relational schema S to XML and $V(I)$ denotes the XML output of this function applied on an instance I of S. Every public query Q has a single free variable, `$CanonicalView`, which is bound to the canonical XML representation of the relational database. We let CV denote the canonical view forest that describes the mapping from a relational schema S to its canonical XML representation. The expression $Q(CV(I))$ denotes the XML output, which is constructed by first evaluating the expression $CV(I)$, then applying Q to the XML result of $CV(I)$, where I is an instance of S. Given V and Q, the algorithm VFCA constructs the view forest $vfca(V, Q)$, denoted V_Q, such that the results of $Q(CV(I))$ and $V_Q(I)$ are identical on any instance I of S, i.e., $Q(CV(I))) = V_Q(I)$ for any I of S. More generally, the VFCA is defined as:

ALGORITHM 1 VIEW FOREST COMPOSITION, VFCA.
Input: *An XQueryCore expression Q over input variables $X_1, \ldots, X_m$ and a view forest V_i corresponding to each input variable X_i.*
Output: *A view forest V_Q =VFCA$(V_1, \ldots, V_m, Q)$ with the property that $Q(V_1(I), \ldots, V_m(I)) = V_Q(I)$ for any I of S.*

There are two applications of the VFCA:

—**Representation of public XML views.** We let Q be the public query with `$CanonicalView` its unique variable, and CV the corresponding canonical view tree for the relational database. The VFCA returns PV =VFCA(CV, Q), which satisfies $Q(CV(I)) = PV(I)$.

—**Query composition.** We let Q be the application query, and V be the view forest for an XML view (V may be the public view forest or the result another composition). The VFCA returns AV =VFCA(Q, V), which satisfies $Q(V(I)) = AV(I)$.

4.1 XQueryCore

The XQuery language provides many features that make queries simpler to write and use, but are also redundant. For instance, complex `FLWR` expressions can be rewritten as the composition of individual `for`, `let`, and `if-then-else` expressions. The XQuery Formal Semantics [World-Wide Web Consortium 2002c] defines a proper subset of the XQuery language, called the XQuery Core language, and gives rules that rewrite or *normalize* every XQuery expression as a XQuery Core expression. The static (type) and dynamic (value) semantics of XQuery is defined on this core language. SilkRoute's XQueryCore language is a proper subset of the XQuery Core language and is defined in Figure 13. It excludes recursive functions and operators (e.g., the "before" and "after" operators (<<//>>) and the `is`/`isnot` operators) and functions that depend on the XML document order. Any public or application query that can be normalized as an XQueryCore expression is supported by SilkRoute.

We assume that XQuery's normalization rules are applied before the VFCA is applied. Many of XQuery's expressions (e.g., predicates in path expressions and comparison operators) have an implicit existential semantics, so many of the normalization rules translate these expressions into equivalent, explicitly quantified

Expr	::=	*Literal*	
	|	`element` *QName* { *Expr* }	Element constructor
	|	`attribute` *QName* { *Expr* }	Attribute constructor
	|	`()`	Empty sequence
	|	*Expr1* `,` *Expr2*	Sequence constructor
	|	*Var*	
	|	*Expr1 BinOp Expr2*	
	|	*UnaryOp Expr*	
	|	`if` (*Expr1*) `then` *Expr2* `else` *Expr3*	
	|	`for` *Var* `in` *Expr2* `return` *Expr2*	
	|	`let` *Var* `:=` *Expr1* `return` *Expr2*	
	|	*QName*(*Expr1, ..., ExprN*)	Function application
	|	*Var* `/` *Axis*`::`*NodeTest*	Single-step path expression
Literal	::=	*String* |*Integer* |*Float* | ...	
UnaryOp	::=	`+` | `-` | `not`	
BinOp	::=	*EqOp* | *ArithOp* | *SetOp* | *LogicalOp*	
EqOp	::=	`eq` | `lt` | `le` | `gt` | `ge` | `ne`	
ArithOp	::=	`+` | `-` | `*` | `div` | `mod`	
LogicalOp	::=	`and` | `or`	
SetOp	::=	`union`	Node set operator
Axis	::=	`self` | `child` | `descendant-or-self` | `parent` | `ancestor` | `descendant`	
NodeTest	::=	*QName* | `*` | `attribute()` | `node()` | `text()`	

Supported functions: Built-in functions `data, count, avg, min, max, sum, distinct-values, empty` and non-recursive, user-defined XQuery functions.

Fig. 13. XQueryCore: A subset of XQuery's normalized core grammar

Core expressions. The normalization rules also include: rewriting literal XML element and attribute constructors into a simpler, non-XML syntax; rewriting `flwr` expressions into nested `for`, `let`, and `if-then-else` expressions in which each `for` and `let` expression binds one variable; and applying the `data` function to element or attribute operands of expressions that require atomic values as arguments. The XQuery Formal Semantics defines in detail all the normalization rules, so we do not enumerate them here.

SilkRoute applies three more normalization rules to path expressions. A path expression containing multiple steps is rewritten into nested `for` expressions, each of which bind one new variable to a single-step path expression. For example, for a multi-step path expression of the form:

`for` *Var* `in` *Expr*`/`*Axis1::NodeTest1*`/.../`*AxisN::NodeTestN* `return` *Expr*

the expression above is rewritten as:

`for` *Var1* `in` *Expr*`/`*Axis1::NodeTest1* `return`
`(... for` *Var* `in` *Var(N-1)*`/`*AxisN::NodeTestN* `return` *Expr*`)`

Disjunctions are eliminated by distributing the `|` (`union`) operator. For example, for the expression:

`for` *Var2* `in` *Var1*`/(`*Expr1* `|` *Expr2*`)` `return` *Expr3*

the expression above is rewritten into:

`(for` *Var2* `in` *Var1*`/`*Expr1*`)` `return` *Expr3*`)` `union`

```
 1. element supplier {
 2.   element company { "Acme Clothing" },
 3.   for $t1 in $CanonicalView/child::Clothing return
 4.     for $c in $t1/child::Tuple return
 5.       for $cat in $c/child::category return
 6.         if (data($cat) eq "outerwear") then
 7.           element product {
 8.             element name { for $item in $c/child::item return data($item) },
 9.             element category { data($cat) },
10.             element description { for $desc in $c/child::description
                                      return data($desc) },
11.             element retail { for $price in $c/child::price
                                 return data($price) },
12.             (for $t2 in $CanonicalView/child::Discount return
13.                for $d in $t2/child::Tuple return
14.                  for $spid in $d/child::pid return
15.                    for $cpid in $c/child::pid
16.                      if (data($spid) eq data($cpid)) then
17.                        element sale {
18.                          for $discount in $s/child::discount return
19.                          for $price in $c/child::price return
20.                            data($discount) * data($price) }
21.                      else ()),
22.             (for $t3 in $CanonicalView/child::Problems return
23.               for $p in $t3/child::Tuple return
24.                 for $ppid in $p/child::pid return
25.                   for $cpid in $c/child::pid return
26.                     if (data($ppid) eq data($cpid)) return
27.                       element report {
29.                          attribute code { for $code in $p/child::code
                                              return data($pcode) },
29.                          for $comments in $p/child::comments return $comments }
30.                       }
31.                     else ())
32.           }
33.         else ()
34. }
```

Fig. 14. NQ_P : Normalization of public query Q_P in Figure 6

(`for` *Var2* `in` *Var1*/*Expr2* `return` *Expr3*)

Lastly, we introduce new variables so that every path expression is rooted with a variable. For example, the expression *Expr1*/*Expr2* is rewritten into: `for` *Var1* `in` *Expr1* `return` *Var1*/*Expr2*

The normalization of public query Q_P is given by the query NQ_P in Figure 14. Note that all multi-step path expressions are rewritten as single-step path expressions and that all path expressions that do not already occur in a `for` expression are replaced with new variables and those variables are bound in a new `for` expression.

4.2 View-Forest Composition Algorithm VFCA

The VFCA is defined recursively on the structure of a XQueryCore expression. The algorithm uses the types and functions in Tables I and II. We note that in the

Expr	Abstract syntax tree of an XQueryCore expression in Figure 13
Node = { QName, Forest, SQL }	Node in a view forest (three-part record)
Forest	Sequence of view-forest nodes
SQL = { SELECT, FROM, WHERE }	SQL fragment (three-part record)
Env = Var → Forest	Environment from a XQueryCore variable to a view forest.

Table I. VFCA Types

`elementNode` : QName × Forest × SQL → Node	Construct element view-forest node
`attributeNode` : QName × Forest × SQL → Node	Construct attribute view-forest node
`atomicNode` : SQL → Node	Construct atomic-value node
`getBinding` : Env × Var → Forest	Return binding of variable in environment
`addBinding` : Env × Var × Forest → Env	Extend environment with binding of variable to view forest
`joinSQL` : $SQL_1 \times \ldots \times SQL_N \rightarrow SQL$	Natural join of N SQL fragments
`forestJoin` : Forest × SQL → Forest	Applies `joinSQL` to SQL fragment of each root in forest and returns new forest
`unionSQL` : $SQL_1 \times \ldots \times SQL_N \rightarrow SQL$	Construct SQL UNION of N SQL queries
`renameTupleVars` : Node → Node	Renames new tuple variables in FROM clauses of Node and its descendants
`renameTupleVarsExceptRoot` : Node → Node	Renames new tuple variables in FROM clauses of Node's children and descendants

Table II. VFCA Functions

XQuery data model [World-Wide Web Consortium 2002a], elements and attributes may contain sequences of atomic values. A node's atomic values are accessed by the `data` function. The XPath 1.0 `text()` accessor returns the lexical representation of a node's content. The VFCA is concerned only with atomic values.

We present VFCA in a functional notation similar to ML [Milner et al. 1990]. The function VFCA(ENV, Q, S) takes three arguments: ENV, an environment that binds XQueryCore variables to view forests; Q, the XQueryCore expression to be converted into a view forest; and S, a SQL fragment, explained below. The VFCA returns a view forest representing Q. The key observation is that all path expressions that occur in XQueryCore expressions can be evaluated on a view forest directly and produce bindings of XQueryCore variables to nodes in the view forest. For example, lines 3–5 of NQ_P in Figure 14 bind variables `$t1`, `$c`, and `$cat` to the nodes $N1$, $N1.1$ and $N1.1.3$ of CV in Figure 10, respectively. In general, VFCA(ENV, Q, S) simulates an evaluation of the XQueryCore expressions, but on view forests instead of XML forests. The main difficulty is computing correctly the new SQL queries on the returned view forest.

The parameter S to VFCA is a SQL fragment and requires some explanation. It is generated for a `for` clause, and is immediately consumed at the root of the corresponding `return` clause. In all other uses in the algorithm, S is empty[3]. It should be interpreted as saying "repeat the entire view forest once for each answer in S". To justify the need for S, consider the following view tree:

```
N1(<product>)  :- FROM Clothing c WHERE c.category = "outerwear"
```

[3]There is one exception, when translating a conditional.

```
N1.1(<report>) :- FROM Problems p WHERE p.pid = c.pid
```

and construct the translations of the two XQueryCore expressions, `Q` and `R`:

```
Q:                                  R:
1. for $x in $view/self::product    5. let $y = $view/self::product
2. return <result>                  6. return <result>
3.          $x                      7.          $y
4.        </result>                 8.        </result>
```

When translating `Q`, VFCA binds the variable `$x` to the view forest:

```
X1(<product>)  :- FROM ()
X1.1(<report>) :- FROM Problems p WHERE p.pid = c.pid
```

This is the original view forest stripped of its SQL query at the root. VFCA also constructs the following query S:

```
S = FROM Clothing c WHERE c.category = "outerwear"
```

and uses it as argument to VFCA when processing line 2. After the view forest for the RETURN clause is computed, S is added to its root. The final view forest that we obtain for `Q` creates one `result` for each product:

```
Q1(<result>)     :- FROM Clothing c WHERE c.category = "outerwear"
Q1.1(<product>)  :- FROM ()
Q1.1.1(<report>) :- FROM Problems p WHERE p.pid = c.pid
```

By contrast, when `R` is translated, the variable `$y` is bound to an isomorphic copy of the original view forest:

```
Y1(<product>)  :- FROM Clothing c WHERE c.category = "outerwear"
Y1.1(<report>) :- FROM Problems p WHERE p.pid = c.pid
```

and the query S is empty when VFCA reaches line 6. As a consequence the view forest resulting for `R` creates one `result` element which contains all the `product` elements:

```
R1(<result>)     :- FROM ()
R1.1(<product>)  :- FROM Clothing c WHERE c.category = "outerwear"
R1.1.1(<report>) :- FROM Problems p WHERE p.pid = c.pid
```

The remainder of this section gives the recursive definition of VFCA. We begin with translation of some simple XQueryCore expressions.

Literal expressions. A literal expression is represented by an atomic-value node whose SQL fragment is a SELECT clause containing the literal value and whose FROM and WHERE clauses are those in `S`.

```
VFCA(Env, [ Literal ], S) =
  let sqlfrag.SELECT = Literal "AS AtomicValue",
      sqlfrag.FROM   = S.FROM,
      sqlfrag.WHERE  = S.WHERE
  in atomicNode(sqlfrag)
```

The SQL attribute `AtomicValue` always contains an atomic value. For XQuery literal expressions, this attribute contains a literal constant.

Element and attribute constructors. An element constructor in Q becomes a new element node in the view forest with the given *QName* and children nodes computed by applying VFCA recursively to *Expr*:

```
VFCA(Env, [ element QName { Expr } ], S) =
  let vf = VFCA(Env, Expr, ()) in elementNode(QName, vf, S)
```

The new node is labeled with the SQL fragment S. When translating the sub-expressions in *Expr*, the new SQL query is the empty SQL fragment (), because all nodes in a view forest implicitly contain the SQL fragments of their ancestors. To illustrate, consider the expression:

```
1. for $p in $view/self::product return
2.   <result> <subresult/> </result>
```

which constructs one `<result>` element for each `<product>`. Assume that `$view` is bound to the view forest:

```
N1(<product>)  :- FROM Clothing c WHERE c.category = "outerwear"
```

When the algorithm reaches line 2, the SQL fragment `S` is the SQL fragment of `N1`. The algorithm constructs a new view-forest node with this SQL query:

```
N1'(<result>) :- FROM Clothing c WHERE c.category = "outerwear"
```

When the algorithm is called recursively on the expression `<subresult/>`, the SQL query is empty:

```
S = FROM () WHERE ()
```

Hence, the complete output view forest is:

```
N1'(<result>)    :- FROM Clothing c WHERE c.category = "outerwear"
N1.1'(<subresult>) :- FROM () WHERE ()
```

The definition of the `attribute` constructor is similar to that for the `element` constructor and is omitted.

Sequence expressions. The empty sequence is translated to the empty view forest:

```
VFCA(Env, [ () ], S) = ()
```

To translate the sequence expression (*Expr1*, *Expr2*), we apply VFCA to each sub-expression, which produces the view forests `vf1` and `vf2`, and then construct the view forest that are the nodes in `vf1` followed by the nodes in `vf2`.

```
VFCA(Env, [ (Expr1, Expr2) ], S) =
  let vf1 = VFCA(Env, Expr1, S),
      vf2 = VFCA(Env, Expr2, S)
  in (vf1, vf2)
```

Now that we have explained the pseudo-code notation, we define the more complex XQueryCore expressions.

Variables. A variable is translated to the view forest `vf` to which it is bound in the current environment. Any subsequent use of *Var* in an XQueryCore expression must include the SQL fragment `S`. Therefore, we "join" `S` with the SQL fragments associated with `vf` by applying the `forestJoin` function:

```
VFCA(Env, [ Var ], S) =
  let vf = getBinding(Env, Var) in forestJoin(vf, S)
```

For each root node `vn` (in `vf`) with SQL fragment `S'`, `forestJoin(vf, S)` computes `S'' = joinSQL(S, S')` and returns a new node `vn'` labeled with SQL fragment `S''`. The function `joinSQL(S, S')` computes a new FROM-WHERE fragment by taking the union of the FROM clauses and the union of the WHERE clauses in `S` and `S'`, i.e., a natural join. To illustrate, consider this query:

```
1. for $p in $view/self::product return
2.   for $r in $p/child::report return
3.     $p
```

and assume that `$view` is bound to the view forest:

```
N1(<product>)  :- FROM Clothing c WHERE c.category = "outerwear"
N1.1(<report>) :- FROM Problems p WHERE p.pid = c.pid
```

The result of this query contains one copy of a `product` element for each report in the product. After line 1, the SQL fragment `S` is:

```
S = FROM Clothing c1 WHERE c1.category = "outerwear"
```

After line 2, the SQL fragment `S` is:

```
S = FROM Clothing c1, Problems p2
    WHERE c1.category = "outerwear" AND p2.pid = c1.pid
```

At line 3, `$p` is bound to the node `N1`. Logically, we need to make a copy of the view forest bound to `$p` and join the SQL fragments associated with its roots with `S`. Assume `$p` is bound to this view forest:

```
P1(<product>)  :- ()
P1.1(<report>) :- FROM Problems p1 WHERE p1.pid = c1.pid
```

Note that the tuple variables in `P1.1` have been renamed. This renaming is necessary to maintain the correspondence between XQueryCore variables and distinct nodes in the view forest. We explain this in detail shortly.

The function `forestJoin` takes the view forest with root `P1` and joins its SQL fragment with `S`, resulting in the following view forest:

```
Q1(<product>) :- FROM Clothing c1, Problems p2
                 WHERE c1.category = "outerwear" and
                       p2.pid = c1.pid
Q1.1(<report>):- FROM Problems p1 WHERE p1.pid = c1.pid
```

Note that this view forest exactly captures the semantics of the XQuery expression above.

Binary operators. The XQuery normalization rules rewrite arithmetic expressions such that a binary operator *ArithOp* is applied to single atomic values, not sequences. As a consequence, after applying VFCA recursively to the two subexpressions, *Expr1* and *Expr2*, each resulting view forest will be one atomic-value node with a SELECT-FROM-WHERE SQL query. The result view tree is another atomic-value node whose SELECT clause computes the arithmetic expression:

```
VFCA(Env, [ Expr1 ArithOp Expr2 ], S) =
  let vfnode1 = VFCA(Env, Expr1, ()),
      vfnode2 = VFCA(Env, Expr2, ()),
      sqlfrag = joinSQL(vfnode1.SQL, vfnode2.SQL),
      sqlfrag.SELECT = vfnode1.SQL.SELECT ArithOp
                       vfnode2.SQL.SELECT "AS AtomicValue"
      resultNode = atomicNode(sqlfrag)
    in forestJoin(resultNode, S)
```

Here `vfnode1.SQL.SELECT` returns the expression in the SELECT field of `vfnode1`, stripped of any `AS` modifiers. Usually the FROM and WHERE clauses of both `vfnode1` and `vfnode2` are empty, because these SQL fragments must return a single value, in their context. However it is possible that they contain one or more joins on foreign keys. Hence we construct the FROM-WHERE clauses in the result by taking their natural join.

For logical operators, we temporarily construct a view forest node with a SELECT-FROM-WHERE clause where the SELECT clause contains the logical expression (a comparison, or a boolean combination of other expressions). Later, in a conditional expression, we move this expression from the SELECT clause to the WHERE clause.

```
VFCA(Env, [ Expr1 (EqOp|LogicalOp) Expr2 ], S) =
  let vfnode1 = VFCA(Env, Expr1, ()),
      vfnode2 = VFCA(Env, Expr2, ()),
      sqlfrag = joinSQL(vfnode1.SQL, vfnode2.SQL),
      sqlfrag.SELECT = vfnode1.SQL.SELECT (EqOp|LogicalOp)
                       vfnode2.SQL.SELECT "AS AtomicValue"
      resultNode = atomicNode(sqlfrag)
    in forestJoin(resultNode, S)
```

This SQL fragment is incorporated into the SQL fragment of the conditional `if` expression in which comparison expressions typically occur. When applying VFCA recursively to the expression in Figure 15 on line 5, we compute the SQL fragment for expressions `data($retail)` and for `data($wholesale) * 2.0`, which are:

```
SELECT p.retail AS AtomicValue
```

and

```
SELECT p.wholesale * 2.0 AS AtomicValue.
```

We combine these fragments into the fragment:

```
SELECT (p.retail > p.wholesale * 2.0) AS AtomicValue.
```

The evaluation of unary operators is similar and is omitted.

```
1. for $p in $CanonicalView/Prices return
2.  for $t in $p/Tuple return
3.    for $retail in $t/retail return
4.      for $wholesale in $t/wholesale return
5.        if (data($retail) > data($wholesale) * 2.0) then
6.          <expensive/>
7.        else <cheap/>
```

Fig. 15. Example use of XQueryCore `if-then-else` expression

Conditional expressions. The boolean expression in an `if-then-else` expression is guaranteed to be to an atomic-value node. To translate an `if-then-else` expression, we first construct the two SQL fragments, `sqlTrue` and `sqlFalse`, which represent the predicate expression *Expr1* and its negation, and combine each fragment with `S`. The result is a sequence of the view forests `vf2` and `vf3`:

```
VFCA(Env, [ if (Expr1) then Expr2 else Expr3 ], S)
  let vn = VFCA(Env, Expr1, S),
      sqlTrue.FROM = vn.SQL.FROM
      sqlTrue.WHERE = vn.SQL.WHERE,"AND (",vn.SQL.SELECT,")"
      sqlFalse.FROM = vn.SQL.FROM
      sqlFalse.WHERE = vn.SQL.WHERE,"AND NOT (",vn.SQL.SELECT,")"
      vf2 = VFCA(Env, Expr2, sqlTrue),
      vf3 = VFCA(Env, Expr3, sqlFalse)
      resultForest = (vf2, vf3)
    in forestJoin(resultForest, S)
```

Recall that `vn.SQL.SELECT` returns the expression in the SELECT clause of `vn`, stripped of any `AS` modifiers. This expression is a boolean expression, which we add to the WHERE clause with an `and` operator. Then we translate the *Expr2* and *Expr3* expressions making sure they are joined with the `sqlTrue` and `sqlFalse` queries respectively. The result consists of a view forest with several possible roots. When the `if-then-else` expression is the result of normalizing an XQuery `where` expression, then the `else` clause is empty, *Expr3* is the empty forest, and `resultForest` consists only of `vf2`.

For example, on lines 5 and 7 in Figure 15, the result of the `if-then-else` is a view forest containing two nodes and whose simplified SQL fragments are:

```
N1.1(<expensive/>) :- WHERE t.retail > t.wholesale * 2.0
N1.2(<cheap/>)     :- WHERE NOT(t.retail > t.wholesale * 2.0)
```

The `for` and `let` expressions. During normalization, each `for` expression is rewritten into:

`for` *Var2* `in` *Var1*/*Axis*::*NodeTest* `return` *Expr*

VFCA on `for` expressions is defined in Figure 16. Lines 3–4 in Figure 16 evaluate the one-step XPath expression on each node in the view forest `vf1` bound to *Var1*. Assume for a moment that *Var1* is bound to a tree. Let `vn1` denote the tree's root and let `vn2` be a node in the given *Axis* that satisfies *NodeTest*. To maintain the dependency between `vn1` and `vn2`, we bind *Var2* to a *copy* of `vn2`, rename some of

```
VFCA(Env, [ for Var2 in Var1/Axis::NodeTest return Expr ], S) =
 1.  let resultVF = (),                  /* The empty forest */
 2.    vf1 = getBinding(Env, Var1) in
        /* Bind vn1 to each root in forest vf1 */
 3.    (for vn1 in vf1 do
          /* Evaluate XPath step on the input view-tree */
 4.       for vn2 in vn1/Axis::NodeTest do
 5.         if (vn2 is vn1 or vn2 is proper ancestor of vn1)
 6.         then let
            /* Copy the tree rooted at vn2,
               rename tuple variables in vn2's children */
 7.                vn2' = renameTupleVarsExceptRoot(vn2),
 8.                S'   = joinSQL(S, vn2.SQL),
 9.                Env' = addBinding(Env, Var2, vn2'),
10.                resultVF := (resultVF, VFCA(Env', Expr, S')))
11.         else /* here vn2 is proper descendant of vn1 */
12.             let [n1, ..., nm] = nodes from vn1 to vn2 (excluding vn1 & vn2),
            /* Copy the tree rooted at vn2, rename tuple variables in vn2 */
13.                vn2' = renameTupleVars(vn2),
14.                S'   = joinSQL(S, n1.SQL, ..., nm.SQL, vn2'.SQL),
15.                vn2''= elementNode(vn2'.QName,vn2'.Forest,()),
16.                Env' = addBinding(Env, Var2, vn2''),
17.                resultVF := (resultVF, VFCA(Env', Expr, S')),
18.   in resultVF
```

Fig. 16. Definition of VFCA for `for` expressions

`vn2`'s tuple variables, and compute the SQL fragment `S'` that depends on the SQL fragments associated with `vn1` and `vn2`. The details of these two steps depend on the relationship between `vn1` and `vn2`.

In the first block (lines 5–10), `v2` is the same node as `vn1` or is an ancestor of `vn1`. We make a copy of `vn2` in which new tuple variables introduced in the FROM clauses of `vn2`'s *children* are renamed (line 7). This guarantees that the correspondence between `vn1`'s tuple variables and those of its ancestor node `vn2` are maintained. The children's variables are renamed so this copy of `vn2` does not conflict with other copies (i.e., other variable bindings). The SQL fragment `S'` is simply the natural join of `vn2`'s SQL fragment with `S` (line 8). The environment `Env'` extends `Env` by binding *Var2* to the new node `vn2'` (line 9), and VFCA is called recursively on *Expr* (the body of the `for` expression) with the new arguments.

In the second block (lines 11–17), `vn2` is a proper descendant of `vn1`. We make a copy of `vn2` in which `vn2`'s new tuple variables are renamed (line 13). In this case, `vn2`'s SQL fragment depends on the SQL fragments of all its ancestors up to `vn1`, therefore `S'` is the natural join of all SQL fragments between nodes `vn1` and `vn2`, excluding `vn1`, but including `vn2`. Because `vn2`'s SQL fragments are included in `S'`, environment `Env'` is extended `Env` by binding *Var2* to a copy of `vn2'` (line 15) with the empty SQL fragment. Subsequent uses of *Var2* in *Expr* will inherit `vn2`'s SQL fragments from `S'`.

The result of the entire `for` expression is a new view forest, which is the sequence of all view forests computed for each node `vn2` by calling VFCA recursively on *Expr* (lines 10 and 17).

To illustrate, assume `$view` is bound to the the view tree:

```
N1(<product>)    :- FROM Clothing c WHERE c.category = "outerwear"
N1.1(<report>)   :- FROM Problems p WHERE p.pid = c.pid
N1.1.1(<comment>):- FROM Comments x WHERE x.cid = p.cid
```

and occurs in the expression:

```
1.  for $p in $view/self::product return
2.    for $r1 in $p/child::report return
3.      for $r2 in $p/child::report return
4.        Expr
```

At line 2 above, the XQuery variable `$p` is bound to `N1`, the node variables `vn1` and `vn2` are bound to `N1` and `N1.1`, respectively, and the sequence `[n1, ..., nm]` is empty. Since `vn2` is a descendant of `vn1`, we rename all tuple variables in the subtree `vn2`, and bind the variable `$r1` to the result. Then at line 14 in VFCA, `$p` is bound to the node `P1`:

```
P1(<product>)     :- ()
P1.1(<report>)    :- FROM Problems p1 WHERE p1.pid = c.pid
P1.1.1(<comment>) :- FROM Comments x1 WHERE x1.cid = p1.cid
```

and `$r1` is bound to the node `R1.1`:

```
R1.1(<report>)    :- ()
R1.1.1(<comment>) :- FROM Comments x2 WHERE x2.cid = p2.cid
```

Notice that the tuple variables `p, x` have been renamed to `p1, x1`, but `c` is not renamed, because it is not defined in this view tree. At line 3 in the expression above, `$r2` is bound to the following view tree:

```
T1.1(<report>)    :- ()
T1.1.1(<comment>) :- FROM Comments x3 WHERE x3.cid = p3.cid
```

This example makes clear the purpose of variable renaming. If the XQuery expression were evaluated on an XML document, then the variables `$r1` and `$r2` would be bound independently, possibly to different nodes. If we did not rename the tuple variable `p` in the nodes `R1.1` and `T1.1`, the resulting SQL fragments would correspond to the case when `$r1` and `$r2` were bound to the *same* `<report>` node. We also rename all new tuple variables in the subtree rooted at `R1.1` and `T1.1`, because the body of the `for` expression may further iterate over subtrees of `$r1` and `$r2`. For example, the two instances of `x` in `R1.1.1` and `T1.1.1` are renamed to ensure that they are distinct.

A `let` expression is much simpler. We simply evaluate the `let` expression and bind the variable to the resulting forest.

```
VFCA(Env, [ let Var1 := Expr1 return Expr2 ] , S)
  let vf1 = VFCA(Env, Expr1, ()),
      Env1 = addBinding(Env, Var1, vf1) in
  VFCA(Env1, Expr2, S)
```

Note that we apply VFCA to *Expr1* with an empty SQL fragment. We do this because *Var1* can only occur in the `let`'s `return` expression *Expr2*. Therefore, `vf1` will be joined with `S` when occurrences of *Var1* in *Expr2* are translated recursively.

Built-in functions and operators. The built-in functions we describe are `data`, `distinct-values`, a generic aggregation function, *agg*, and `empty`. Because we rely on type-correct expressions, the view forest `vf` argument to `data` is a single node that contains exactly one child, which is a atomic-value node; `vf/child::*` returns the forest consisting of all these children.

```
VFCA(Env, [ data(Expr) ], S) =
  let vf = VFCA(Env, Expr, S) in vf/child::*
```

The evaluation of `distinct-values` requires a nested SQL query. The argument to `distinct-values` is guaranteed to be a forest of atomic-value nodes. For each node in `vf`, we compute the SELECT DISTINCT sub-query and take the union of all these queries. We rename the atomic value of each SQL fragment as attribute `AtomicValue`. Recall that a SQL union eliminates duplicates. Finally, we create a top-level SELECT-FROM-WHERE query, as a place-holder, because every node in the view forest must have a query of this form.

```
VFCA(Env, [ distinct-values(Expr) ], S) =
      /* a forest of atomic value nodes */
  let vf = VFCA(Env, Expr, S),
      sqlfrag := ()
      (for vn in vf do
         /* vn is bound to a atomic value node */
         let sqlfrag'.SELECT = "DISTINCT" vn.SQL.SELECT
                               "AS AtomicValue",
             sqlfrag'.FROM   = vn.SQL.FROM,
             sqlfrag'.WHERE  = vn.SQL.WHERE,
             sqlfrag := unionSQL(sqlfrag, sqlfrag')),
      sqlResult.SELECT = "Expr.AtomicValue",
      sqlResult.FROM   = sqlfrag "Expr"
   in atomicNode(sqlResult)
```

As with `distinct-values`, the aggregate functions require a nested SQL query. We take the union of all SQL fragments of all nodes in `vf`. For aggregate functions that take atomic values, we assume that all SQL fragments name their unique atomic value as attribute `AtomicValue`:

```
VFCA(Env, [ agg(Expr) ], S) =
      /* a forest of atomic value nodes */
  let vf = VFCA(Env, Expr, S),
      (vn1, ..., vnm) = vf,  /* all nodes in the forest */
      unionSql = unionSQL(vn1.SQL, vn2.SQL, ..., vnm.SQL),
      sqlResult.SELECT = agg "(Expr.AtomicValue)",
      sqlResult.FROM   = unionSql "Expr"
  in atomicNode(sqlResult)
```

For the `count` aggregate function, the nested sub-query is simply the union of all

SQL fragments:

```
VFCA(Env, [ count(Expr) ], S) =
  let vf = VFCA(Env, Expr, S) /* a forest of atomic value nodes */
      (vn1, ..., vnm) = vf,  /* all nodes in the forest */
      unionSql = unionSQL(vn1.SQL, vn2.SQL, ..., vnm.SQL),
      sqlResult.SELECT = "count (*)"
      sqlResult.FROM   = unionSql
  in atomicNode(sqlResult)
```

For the `empty` function, the nested sub-query computes the union of all SQL fragments associated with *Expr*. The result is an atomic-value node that computes true if the union is empty and false, otherwise.

```
VFCA(Env, [ empty (Expr) ], S) =
  let vf = VFCA(Env, Expr, S)
      (vn1, ..., vnm) = vf,  /* all nodes in the forest */
      unionSql = unionSQL(vn1.SQL, vn2.SQL, ..., vnm.SQL),
      sqlTrue.SELECT = "TRUE",
      sqlTrue.WHERE = "NOT EXISTS(" unionSql ")",
      sqlFalse.SELECT = "FALSE",
      sqlFalse.WHERE = "EXISTS(" unionSql ")",
      sqlResult.FROM = sqlTrue "union" sqlFalse
  in atomicNode(sqlResult)
```

The XQueryCore node-set operator `union` is defined on equality of node identity, i.e., it compares the identity, not the content, of element and attribute nodes in an XML document. By definition, the nodes in a view forest denote disjoint sets of XML values in a materialized XML document, therefore, we simply apply the XQuery `distinct-nodes` operator to the view forest nodes themselves.

```
VFCA(Env, [ Expr1 union Expr2 ], S) =
  let vf1 = VFCA(Env, Expr1, S),
      vf2 = VFCA(Env, Expr2, S),
  in distinct-nodes(vf1, vf2)
```

User-defined functions. The VFCA supports non-recursive, user-defined XQuery functions. The translation of a user-defined function first applies VFCA to each actual function argument, *Expr1* ... *ExprN*, resulting in the view forests `vf1,...,vfn`. The definition of function includes its formal arguments (*Var1* ... *VarN*) and function body (*Expr*). A new environment `Env'` binds each formal argument to its corresponding actual view-forest value, then we apply VFCA recursively to the function body with the new environment and SQL fragment S.

```
VFCA(Env, [ QName (Expr1, ..., ExprN) ], S) =
  let vf1 = VFCA(Env, Expr1, ())
      ...
      vfn = VFCA(Env, ExprN, ())
      /* lookup definition of function QName with
         formal arguments Var1...VarN and function body Expr */
```

```
        Env' := addbinding((), Var1, vf1)
        ...
        Env' := addbinding(Env', VarN, vfn)
    in VFCA(Env', Expr, S)
```

This translation essentially "inlines" the function. This technique cannot be applied to external functions (i.e., those not written in XQuery) because their definitions are not available nor can it be applied to recursive XQuery functions because the resulting a view forest must be a finite tree and a recursive function cannot be expressed as a finite tree.

4.3 Discussion

We summarize with a discussion of the VFCA's significance and applicability.

Significance. The VFCA is an example of *partial evaluation*: path expressions, and the binding of variables in `for` and `let` expressions are evaluated at *compile* time on the input view forests, and conditional `if-then-else` expressions and the element and attribute constructors in `return` expressions are evaluated at *run time* when the result view forest is materialized.

A significant property of VFCA is that it gives a unique, canonical representation of any XQueryCore expression, therefore is applicable to systems other than SilkRoute. In general, XQueryCore expressions can be complex: they can create intermediate XML trees, apply aggregate functions and `distinct-values()`, then iterate over these results to create new XML trees. The actual XML tree that is constructed and returned may be buried deep in the XQueryCore expression. The VFCA transforms such a query into a canonical form, such that (1) only the result XML tree is constructed, (2) all the intermediate XML trees are eliminated, and (3) all the computations that extract data are expressed with SQL queries, not XQueryCore expressions.

Applicability. We omitted from our discussion several XQuery features, such as certain navigation axis and types. Although we described how to translate parent and ancestor axes in the FOR expression, we did not explain how to compute the parent of a new node in a view forest: This should be done according to the XQuery Data Model [World-Wide Web Consortium 2002a] semantics, and we omit it from this paper. Support for navigation axes that depend on document order (e.g., `following/preceding-sibling`) and for recursive functions require extensions to the VFCA.

Making VFCA type-aware is essential for practical applicability. The algorithm is sensitive to type errors, and requires that the input XQueryCore expression be type correct, e.g., it assumes that nodes of type `attribute()` have a unique child of atomic type. Typically, type checking is applied to the XQueryCore query before the composition algorithm. If that is not possible, one can modify the composition algorithm to deal with type errors, for example, by reporting detected type errors or by generating an empty document at run-time.

Finally, we note that the problem of simplifying the resulting SQL expressions is orthogonal to the VFCA. The VFCA tends to build nested SQL statements such as:

```
FROM (SELECT * FROM R x WHERE x.a > 4) y
WHERE y.b < 3
```

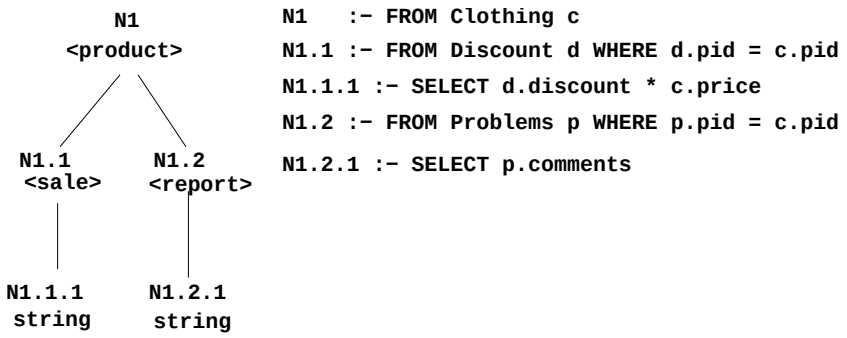


Fig. 17. View forest for query fragment in Figure 6

which can be simplified to:

```
FROM  R y
WHERE y.b < 3 AND  WHERE y.a > 4
```

While some ad-hoc rules for simplification are obvious, care must be taken for subqueries that have the `DISTINCT` keyword. We refer the reader to material [Pirahesh et al. 1997] for a discussion SQL rewriting.

5. EFFICIENT EVALUATION OF VIEW FORESTS

To materialize a view forest as an XML document, SilkRoute's planner takes a view forest and emits one or more SQL queries, which are sent to the relational engine for evaluation. There are two common strategies for translating a view forest into a set of SQL queries. The *fully partitioned* strategy computes one SQL query for each node in the forest and sends all these SQL queries to the relational engine, whereas the *unified* strategy computes one SQL query for the entire view forest. For view forests that represent application queries, either strategy is usually feasible, because the materialized result of an application query is small compared to the relational database. Choosing an efficient evaluation strategy, however, is important when the view query materializes a large fragment of the relational database.

Here, we address the problem of publishing large view forests. There are many possible translations of a view forest into a set of SQL queries. We call such a set of SQL queries a *plan*. Choosing a plan is an optimization problem and the search space consists of all partitions of the view forest. We call each partition a *view-forest decomposition*. SilkRoute constructs a set of SQL queries that correspond to a decomposition and submits the queries to the relational database. We focus here on the problem of finding good plans.

We illustrate the optimization problem using a fragment of the public query contained in the boxes in Figure 6. Figure 17 (left) depicts the view forest for the query fragment. In this case, the view forest is a tree; its structure makes it clear how to generate queries. An edge between a parent and child node requires a left-outer join between the parent's and child's SQL queries. Unions are required to combine the SQL queries of sibling nodes and to combine the SQL queries for each individual view tree in the view forest. We focus on translating single view trees,

```
SELECT 1 AS L1, c.pid, L2, (c.price * Q.discount) as sale, Q.code, Q.comments
FROM Clothing c
LEFT OUTER JOIN
((SELECT 1 AS L2, d.pid AS pid, d.discount AS discount, null AS code,
         null AS comments
  FROM  Discount d)
  UNION
  (SELECT 2 AS L2, p.pid AS pid, null AS discount, p.code AS code,
         p.comments AS comments
    FROM Problems p)
 ) AS Q
 ON c.pid = Q.pid
 ORDER BY L1, c.pid, L2, Q.code
```

Fig. 18. Complete SQL query for view forest in Figure 17.

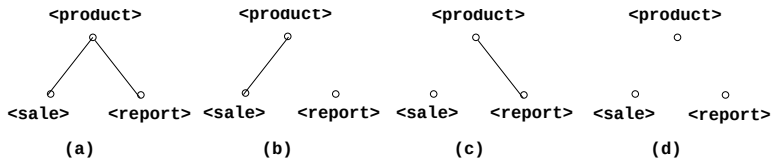


Fig. 19. Execution plans for query fragment

but note that the same techniques apply when the view forest contains multiple view trees.

Figure 18 contains the SQL query that corresponds to the view forest in Figure 17. The outer join is necessary, because there could be products without discounts or reports, and they must appear in the XML document. The `order-by` clause with the attributes `L1`, `c.pid`, `L2`, and `Q.code` groups tuples from the same supplier together in an order that permits the XML generator to construct the `<product>` elements in a single pass over the tuples. This query is a unified plan, because it corresponds to the entire view tree and produces one relation.

The unified plan is not the only choice. We can split the view tree into connected components, and generate a separate SQL query for each such component. Figure 19 illustrates these choices: (a) corresponds to the query above, while (b), (c), and (d) are three alternative ways to partition the view tree into connected components. (We omit the atomic-value nodes here because they are trivially contained in their parents.) Each produces a set of SQL queries. For example, Plan (b) results in the two SQL queries in Figure 20. Notice that the query on the right has no outer join, because the query on the left produces all the values for `product`. The XML generator merges the two sorted tuple streams to produce the XML elements.

In general, there are $2^{|E|}$ possible translations of a query into one or more SQL queries, where $|E|$ is the number of edges in the query's corresponding view tree. Given the exponential number of potential plans, SilkRoute uses heuristics to choose a good plan. In commercial XML middle-ware products, the user typically must write these SQL queries himself, which effectively "hard wires" the evaluation plan into the XML view. This requirement may seem reasonable, but in practice, it is

```
SELECT 1 as L1, c.pid,
       (d.discount * c.price)
FROM Clothing c
LEFT OUTER JOIN
(SELECT d.pid, d.discount
 FROM  Discount d)
ON c.pid = d.pid
ORDER BY c.pid
```

```
SELECT 2 as L1, c.pid, p.code, p.comments

FROM Clothing c, Problems p
WHERE c.pid = p.pid
ORDER BY c.pid, p.code
```

Fig. 20. Complete SQL queries for view forest in Figure 19(b).

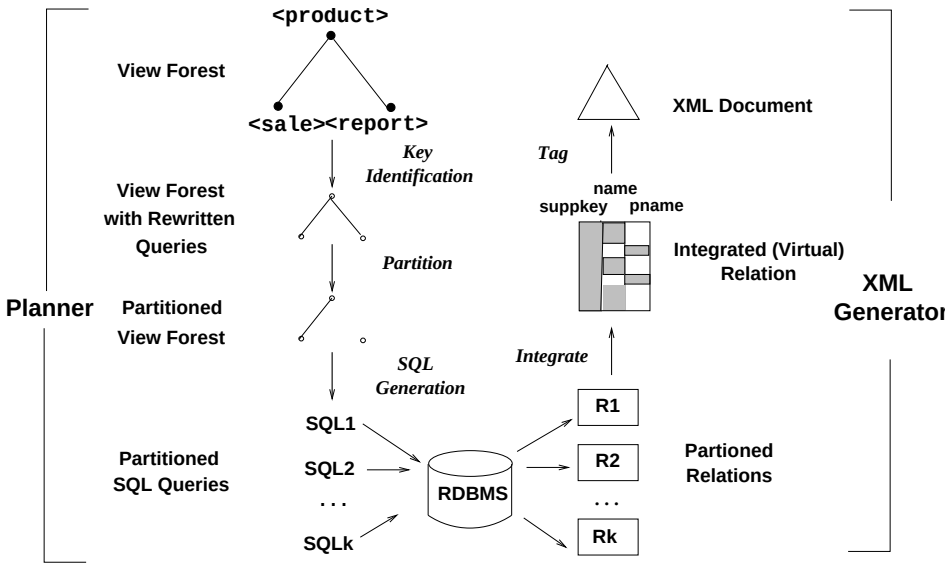


Fig. 21. Architecture of query planner and XML generator

difficult to choose a good plan. The simplest choices are to produce one unified relation as in Figure 19(a) or fully partitioned relations as in Figure 19(d). We show in Section 5.4 that these two plans may be substantially slower than the optimal plans.

5.1 View-Forest Evaluation

Figure 21 depicts the architecture of the planner and XML generator. First, the planner rewrites each SQL fragment by adding a key to each query. Then the planner partitions the view tree into a spanning forest with one or more sub-trees, and for each sub-tree, the planner generates one SQL query. The set of all SQL queries are evaluated by the relational engine, producing a set of tuple streams. The XML generator consumes these tuple streams and constructs one integrated (virtual) relation. A tuple in the integrated relation represents a path from the root element to a leaf value in the result XML document. The XML document is constructed by re-nesting the tuples in the integrated relation and tagging each element. We describe each step in more detail.

Step 1: Key identification. A key is constructed for each query Q in the view tree, by taking the union of the attributes that are explicitly specified in Q's SELECT clause and a set of key attributes. Let the complete SQL query for a node be SELECT A FROM $e_1\ x_1, \ldots, e_n\ x_n$, where A denotes a set of attributes. We replace A with a key $\cup_{i=1}^{n} key(e_i) \cup A$. When e_i is a relation R, $key(e_i)$ is the set of R's key attributes.

Constructing queries with keys is a necessary and sufficient condition for computing tuple streams that can be merged by the XML generator in constant space. It is sufficient, because no other attributes are required for XML view construction, and it is necessary, because without keys, we cannot sort tuples so that the XML generator can construct the XML document in a top-down, single pass over the tuple streams. XQuery's semantics assumes identities of XML elements, so the relational data that instantiates XML views must have tuple identifiers or keys to be consistent with XQuery's semantics.

Step 2: View-tree partitioning. In general, the planner produces one plan for each spanning forest of the view tree, so it produces $2^{|E|}$ plans, where $|E|$ is the number of view-tree edges. In Section 5.5, we present a greedy algorithm that heuristically chooses a subset of the $2^{|E|}$ plans.

For each tree in a spanning forest, we define the schema of the relation that computes the nodes in that tree. Let T_i be one tree in a spanning forest of view tree T, and let *NIDmax*(T_i) be the maximum length of the integer sequences of node identifiers in T_i (e.g., the *NIDmax* of a tree containing the node identifiers `N1.1` and `N1.1.1` is three). Let R_i be the relation that corresponds to T_i. The schema for R_i is the set of attributes $attrs(R_i) = NIDattrs_i \cup Sattrs_i$ where

—$NIDattrs_i = \{\mathtt{L}_j | 1 \leq j \leq NIDmax(T_i)\}$, and
—$Sattrs_i = \{v | v$ is an attribute in SELECT clauses in $T_i\}$.

A tuple t_i in an instance of R_i represents an instance of a path from the root to a leaf node n in T_i, i.e., a path from the document node to a atomic value in the result XML document. The attribute $\mathtt{L}_j$ contains the jth integer of n's identifier, and the attributes in $Sattrs_i$ contain the values necessary to compute an instance of n, i.e., values for attributes in C_n`.SQL.SELECT`. Note that tuple t_i contains enough information to generate all XML elements on the path from the root to n. The tuples are sorted by $\mathtt{L}_1$, $V_{(1,1)}, \ldots, V_{(1,n_1)}$, $\mathtt{L}_2$, $V_{(2,1)}, \ldots, V_{(2,n_2)}$, etc., where $V_{(i,j)}$ denotes a SELECT clause attribute in query fragments of view-tree nodes at level i. This order is consistent with the structural relationship between the elements in the result XML document. We call the set of $\mathtt{L}_i$ and $v_{(i,j)}$ the level-i attributes. If a set of tuples have the same values for all level 1 to level i attributes, then the tuples correspond to a set of leaf values that have the same ancestor elements from level 1 to level i. The level-i attributes permit SilkRoute to merge partitioned relations into an integrated relation in a single pass.

Step 3: Query construction. SilkRoute constructs *outer-join* plans for partitioned relations. The outer-join plans use SQL's outer-join and union operators and the structure of these plans correspond to the structure of trees in a spanning forest. Let n be a node in a tree and let $(n_1, \ldots, n_k)$ be the k children of n. The outer-join plan for n is $OJP(n) =$ n`.SQL` leftjoin $(OJP(n_1) \cup \ldots \cup OJP(n_k))$. The sub-queries for the sibling nodes $n_1, \ldots, n_k$ (n's children) are combined with a left-outer union, and the sub-query for n and the sub-queries of its children are combined with an outer

Relation	A partitioned relation
Tuple	Tuple in the integrated relation $(L_1, V_{(1,1)} \ldots, V_{(1,n_1)}, \ldots L_m, V_{(m,1)} \ldots, V_{(m,n_m)})$
Tag	Set of tags
NID	A node-identifier $(l_1, \ldots, l_m)$
FV	Field values $(v_{(1,1)}, \ldots, v_{(1,n_1)}, \ldots, v_{(m,1)}, \ldots v_{(m,n_m)})$

Table III. Types

`getTuple`: Relation → Tuple	Returns next tuple from integrated relation
`getTag`: NID → QName	Returns QName tag of node-identifier index
`getNodeIdentifier`: Tuple → NID	Projects node-identifier index values from tuple
`getValues`: Tuple → FV	Projects $V_{(i,j)}$ values from tuple
`getLeaf`: Tuple × NID → string | integer | ... | null	Returns atomic value associated with node identifier, or null if it has no atomic value
`SAXWriter`	Implementation of SAX Writer
`emitXML`	Emits tags and values for given tuple
`generateXML`	Given partitioned relations, generates XML document

Table IV. Functions and procedures

join. Note that this query computes instances of the paths in the view tree. The outer union is necessary because sibling nodes have different relational schemas: In the relation that computes a node n_i, the attributes of $n_j (j \neq i)$'s are null values.

Step 4: XML generation. SilkRoute does not materialize the integrated relation. Instead, the result XML document is constructed directly from the partitioned relations. Figure 22 contains the XML generation algorithm, defined in procedure `generateXML`. The algorithm depends on the types and functions defined in Tables III and IV. Intuitively, it merges multiple tuple streams into one tuple stream, nests the tuples, and tags their values. The required memory size of the algorithm depends only on the size of the view forest. It does *not* depend on the size of the database instance, therefore the algorithm scales well as the size of the underlying database and corresponding XML document increases.

5.2 Example

We illustrate the evaluation procedure on the example in Figure 19.

Step 1: Keys are identified and added to `select` clauses. For example, `c.pid` is added to the select clauses of the queries for N1.1.1 and N1.2.1. The complete queries for N1.1.1 and N1.2.1 are, respectively:

```
SELECT c.pid, d.pid,
       (d.discount * c.price) AS sale
FROM Clothing c, Discount d
WHERE c.pid = d.pid
```

```
SELECT c.pid, p.pid, p.code,
       p.comments
FROM Clothing c, Problems p
WHERE c.pid = p.pid
```

Step 2: We give the schemas and instances for Figures 19(a) and (b). Given the database instance on the top in Figure 23, the result relation for Figure 19(a) is in Figure 24. This relation corresponds to the un-nested version of the result XML document in Figure 23 (bottom). Figure 19(b) yields the partitioned relations in Figure 25. The relation on the right corresponds to the tree containing only the `report` node, where the (maximum) length of the node identifier N1.2 is two so

```
procedure generateXML(Relations : {Relation}) {
   SAXWriter.startDocument()
   // Initialize all node identifiers and field values to null
   nid' = (L_1 : null, ..., L_m : null)
   values' = (V_(1,1) : null, ..., V_(m,n) : null)
   // Get next tuple from Relations in order
   // (L_1, V_(1,1) ..., V_(1,n_1), ... L_m, V_(m,1) ..., V_(m,n_m))
   while ((tuple = getTuple(Relations)) != EOF) {
      nid = getNodeIdentifier(tuple)
      values = getValues(tuple)
      if (nid' != nid or values' != values) {
         // Get maximum index where new tuple and old tuple differ
         let l_1, ..., l'_m = nid'
             l_1, ..., l_m = nid
             n_1 = max{i|nid.L_i = nid'.L_i},
             n_2 = max{i|values.V_(i,j) = values'.V_(i,j)},
         in emitXML(m', min(n_1, n_2)+1, m, tuple)
      }
      nid' = nid
      values' = values
   }
   SAXWriter.endDocument()
}
procedure emitXML(m_2, n, m_1, tuple) {
   nid = getNID(tuple)
   // Close all open elements up to new element
   for (i = m_2; i ≥ n; i = i − 1)
      SAXWriter.endElement(getTag(nid.L_1 ...nid.L_i))
   // Open all containing elements up to new element
   for (i = n; i ≤ m_1; i = i + 1) {
      SAXWriter.startElement(getTag(nid.L_1, ...,nid.L_i))
      leafValue = getLeaf(nid.L_1, ..., nid.L_i, tuple)
      if (leafValue != null) SAXWriter.characters(leafValue)
   }
}
```

Fig. 22. XML Generation Algorithm

$NIDattrs_1 = \{L_1, L_2\}$ and $Sattrs_1 = \{\texttt{c.pid}, \texttt{p.pid}, \texttt{p.code}, \texttt{p.comments}\}$.

Step 3: Let n be the root node of the viewtree in Figure 19(a). Then, $OJP(n)$ is the SQL query in Figure 18. Let $n1$ be the root node and let $n2$ be the `report` node in the viewtree in Figure 19(b). Then, $OJP(n1)$ is the SQL query in Figure 20 (left) and $OJP(n2)$ is the SQL query in Figure 20(right).

Step 4: The `generateXML` procedure nests and tags XML values by computing the difference between two successive tuples in the integrated relation. When applied to the tuples in Figure 24, the procedure reads in the first tuple and emits the first two lines in Figure 23. The first and second tuples in the relation have the same values in all level-1 attributes (i.e., L_1 and $c.pid_{(1,1)}$), but differ in the other attributes, therefore, the procedure `emitXML` closes the current level-2 tag(`<sale>`), opens the new level-2 tag (`<report>`) in the same level-1 element(`<product>`), and emits the

```
Clothing(pid:c#1, item:"green skirt",.. , price:"50.00", ..)
Clothing(pid:c#2, item:"red kimono",.. , price:"200.00", ..)
Clothing(pid:c#3, item:"yellow T-shirt",.. , price:  "30.00", ..)
Clothing(pid:c#4, item:"blue jacket",.. , price:"70.00", ..)

Discount(pid:c#1, item:"green skirt", discount: 0.2)
Discount(pid:c#2, item:"red kimono", discount: 0.6)
Discount(pid:c#4, item:"blue jacket", discount: 0.8)

Problems(pid:c#1, code:0012, comments:"fits poorly")
Problems(pid:c#1, code:0035, comments:"button missing")
Problems(pid:c#4, code:0004, comments:"zipper jams")
```

```
<product>   <!-- green skirt -->
  <sale>10</sale>
  <report>fits poorly</report>
  <report>button missing</report>
</product>
<product>   <!-- red kimono -->
  <sale>120</sale>
</product>
<product>   <!-- yellow T-shirt -->
</product>
<product>   <!--  blue jacket -->
  <sale>56</sale>
  <report>zipper jams</report>
</product>
```

Fig. 23. Example fragment of the supplier's database instance and fragment of result XML document

L_1	L_2	$c.pid_{(1,1)}$	$sale_{(2,1)}$	$p.code_{(2,2)}$	$p.comment_{(2,3)}$
1	1	c#1	10		
1	2	c#1		0012	"fits poorly"
1	2	c#1		0035	"button missing"
1	1	c#2	120		
1		c#3			
1	1	c#4	56		
1	2	c#4		0004	"zipper jams"

Fig. 24. Integrated relation for Plan (a)

Plan (b), edge `<product>-<sale>`:

L_1	L_2	$c.pid$	$sale$
1	1	c#1	10
1	1	c#2	120
1		c#3	
1	1	c#4	56

Plan (b), node `<report>`;

L_1	L_2	$c.pid$	$p.code$	$p.comment$
1	2	c#1	0012	"fits poorly"
1	2	c#1	0035	"button missing"
1	2	c#4	0004	"zipper jams"

Fig. 25. Relations for Plan (b) in Figure 19

p.comment value. The processing of the third tuple is similar. The third and fourth tuples differ in the level-1 attribute $c.pid_{(1.1)}$, therefore, `emitXML` closes the level-1 `<product>` element, open tags at the level 1 and level 2 elements (`<product>` and `<sale>`), and emits the sale value.

Because the tuples in all relations are sorted in order by L_1, $V_{(1,1)}$, ..., $V_{(1,n_1)}$, ..., L_m, $V_{(m,1)}$, ..., $V_{(m,n_m)}$, the `generateXML` procedure can merge tuples from multiple partitioned relations and construct the integrated relation in a single pass. For example, in the two partitioned relations in Figure 25, the first tuple read is the first tuple in the relation for `<product>-<sale>`, and the second tuple read is the first tuple in the relation for `<report>`.

Discussion. The outer-join plan is different from the outer-union plan [Shanmugasundaram et al. 2000]. The outer-join plan corresponds to (R *leftjoin* ($S \cup T$)) whereas the outer-union plan corresponds to ((R *leftjoin* S) $\cup$ (R *leftjoin* T)). The outer-union plan combines parent and child nodes using inner or outer joins and combines sub-trees with outer unions. In general, SilkRoute can use either strategy to generate queries, but it currently implements outer-join plans. For completeness, we include the outer-union plan in the experiments in Section 5.4 and distinguish clearly between the unified outer-join and outer-union plans in our results.

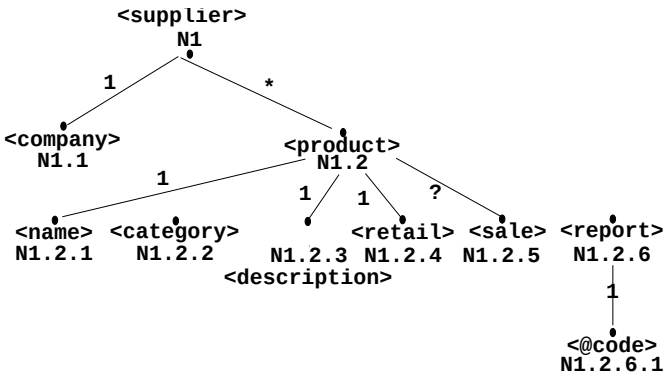
Some plans that SilkRoute produces do not require outer union, outer join, or the with clause. For example, a fully partitioned plan has no edges and requires none of these constructs. Plans with no branches (i.e., no sibling nodes) do not require the union operator. This characteristic is especially useful in a middle-ware system, because all SQL engines do not necessarily support all these constructs. In those cases, SilkRoute chooses permissible plans based on the source description of the relational engine.

5.3 View-forest reduction

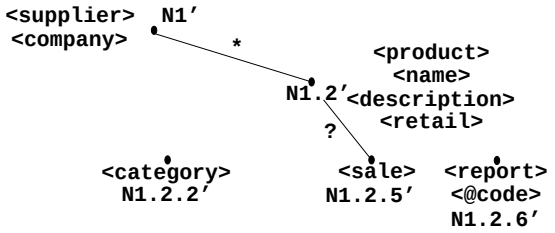
So far, we have assumed that one SQL query is associated with exactly one node in the view forest. We now relax this restriction by allowing an SQL query to be associated with more than one node. We do this by eliminating "reducible" edges in a view forest. An edge between a parent and child node is reducible if the query of the child node has functional and inclusion dependencies on the query of its parent node. When the parent and child queries are equivalent, we can generate XML elements for both nodes from the same tuple. Eliminating reducible edges yields a simpler view forest with fewer SQL queries to evaluate.

After generating a partitioned view tree, the planner reduces the view tree in two steps. First, each edge in the view tree is assigned a label that indicates the potential number of child elements in the XML document. Second, groups of nodes connected by '1'-labeled edges are collapsed into one node by combining their queries. After reduction, SQL generation proceeds as described in Section 5.1.

We illustrate the labeling step on the partitioned view tree in Figure 26. The edge labels '1', '?', '+' and '*', denote one, zero or one, one or more, and zero or more child elements, respectively. An XQuery query does not contain sufficient information to label edges, because the possible number of XML elements depends on the database constraints. The database constraints can be derived from key constraints and referential constraints extracted from the schema of the relational database. Given these inputs, SilkRoute labels the view forest edges as follows. As-



(a) Partitioned view tree of an execution plan



(b) Partioned view tree after reduction

Fig. 26. Example of view forest reduction

sume that p and c are the parent and child nodes of an edge e and that the schema of their complete SQL queries' results are $R_p(x_1, \ldots, x_m)$ and $R_c(x_1, \ldots, x_m, \ldots, x_n)$, respectively. Then, e is labeled:

		C1	
		true	*false*
C2	*true*	1	+
	false	?	*

—**C1** is true if and only if there exists a functional dependency $R_c : x_1, \ldots, x_m \rightarrow x_{m+1}, \ldots, x_n$.

—**C2** is true if and only if there exists an inclusion dependency $R_p[x_1, \ldots, x_m] \subseteq R_c[x_1, \ldots, x_m]$.

The inverse of **C2**, $R_c[x_1, \ldots, x_m] \subseteq R_p[x_1, \ldots, x_m]$, always holds, because XQuery's semantics always define a tree. Therefore, **C2** implies $\pi_{x_1,\ldots,x_m}(R_c) = R_p$ in this context. In general, the problem of checking whether a given set of functional and inclusion dependencies implies another set of dependencies is undecidable [Abiteboul et al. 1995]. SilkRoute uses heuristics and known algorithms for restricted problems. In particular, it does not consider inclusion dependencies when it checks if a functional dependency can be derived, which allows the check to be done in linear time [Beeri and Bernstein 1979]. From our experience, this solution is adequate for typical XQuery queries.

In the second step, the view tree's nodes are grouped into equivalence classes of nodes that are reachable only by '1'-labeled edges. Figure 26 illustrates this step. For each such class, the query fragment of the least-common-ancestor node is replaced by Q whose SELECT clause is the union of all SELECT clauses on the nodes in the class. Q's FROM and WHERE are constructed in the same way. In Figure 26 (a), the equivalence classes are {`N1, N1.1`}, {`N1.2, N1.2.1, N1.2.3, N1.2.4`}, and {`N1.2.6, N1.2.6.1`}. They are replaced by `N1'`, `N1.2'`, and `N1.2.6'`, respectively.

View-forest reduction can reduce the number of outer joins[4] and can reduce the total size of the relations and therefore, the total size of data transferred. The actual impact of view-forest reduction on the data size depends on the characteristics of submitted queries and database instances. For example, in Figure 26, if the data size of the `<company>` element dominates, then in the reduced view forest, its large data value would occur in every tuple in the relation for `N1'`, which could increase data-transfer time. Both query-only time and data-transfer time of a reduced plan, therefore, may not always be faster than the corresponding non-reduced plan. To alleviate this problem, we can prohibit the reduction of specific nodes based on the average data size estimated by the target database. We use view-forest reduction as a plan-improving heuristic: given a set of arbitrary non-reduced plans, the corresponding set of reduced plans, in general, are more efficient. Our experimental results support this heuristic.

5.4 Experiments

A view forest permits us to generate and compare all possible execution plans for a public query. Here, we present experiments that compare the unified and fully partitioned plans to the "optimal" plans, i.e., those plans that have the fastest execution times compared to all others. SilkRoute uses an outer-join strategy to generate plans, so its unified plans are not equivalent to outer-union plans [Shanmugasundaram et al. 2000]. For completeness, we include a unified outer-union plan in the experiments. We also compare plans generated from non-reduced view forests with those generated from reduced view forests.

We use the TPC Benchmark 'H' database [Council 2001], which contains information about parts, the suppliers of those parts, customers, and their part orders. We generated two public queries, $Q1_P$ and $Q2_P$, for the database.[5]. They both specify the entire contents of the TPC database, but are differ in structure. In $Q1_P$, two one-to-many edges (labeled "*"), are nested in a chain, whereas in $Q2_P$, the two '*' edges are parallel. A '*' edge corresponds to a outer join in an SQL query, so each query stresses the relational engine differently: $Q1_P$ has nested outer joins and $Q2_P$ has unions of outer joins. Each view tree for $Q1_P$ and $Q2_P$ has nine edges (ten nodes). As described in earlier, one plan is generated for each subset of edges in the view tree, so there are 512 plans for each query. Each plan generates between one and ten SQL queries, each of which produces one tuple stream.

The experiments were run using the two database configurations in Table V. Configuration A used the TPC-H Database with 1 MB of data, and Configuration

[4]The current query generator constructs for each node an outer join with the union of its children, which disappears when all children are labeled '1'.

[5]The queries $Q1_P$ and $Q2_P$ and their view forests are given in Appendix A

Config	Size	Database Server Platform	Client Platform
A	1 MB	AMD K6-2 350 MHz 256MB mem 1GB swap Linux RH 6.1	SGI Challenge L 4GB mem IRIX64 V6.5
B	100 MB	Intel Celeron 566 MHz 256 MB mem 1GB swap Linux RH 6.2	Intel Pentium III 192 MB mem Linux RH 6.2

Table V. Experimental Configurations

B used a 100 MB database. Exhaustive query plans were generated for Configuration A; Configuration B is used in Section 5.5 to evaluate our plan-generation algorithm. Due to licensing restrictions, we are not permitted to identify the commercial product used in our experiments. In the experiments, the database client was a simple Java program that submitted SQL queries to the database server and read tuples from the tuple streams via JDBC. Both configurations use JDK 1.2 and JDBC 1.2.2.

Figures 27 and 28 plot the execution times of the 512 plans for $Q1_P$ and $Q2_P$, respectively. The horizontal axis is the number of tuple streams per plan and the vertical axis is the execution time in milliseconds, on a log scale. Both total time and query-only time were measured on the SilkRoute client. Total time includes query execution time on the database server and data transfer time to the client: timing began when the first SQL query was submitted to the server and terminated when the *last* tuple was read from the last tuple stream. Query-only time includes the time until the *first* tuple is read from a tuple stream. The time to first tuple is comparable to the time to count all tuples in the result on the server only, so pipelining of output during query execution did not affect our measurements. If a sub-query did not complete within 5 minutes, no time was reported. For $Q1_P$, 101 plans timed out; for $Q2_P$, no plans timed out. Each plan was executed twice successively, and we took the latter result. We believe that the caching effect is negligible, because independent executions of sample queries showed little difference.

For non-reduced trees, the outer-union and fully partitioned plans are slightly slower than the optimal plans. Figures 27(a) and 28(a) plot the query-only time for non-reduced trees. In Figure 27(a), the unified outer-union plan (diamond) is 16% slower than optimal and the fully partitioned plan is 24% slower. In Figure 28(a), the outer-union plan is 21% slower and the fully partitioned plan is 41% slower.

Recall from Section 5.3 that view-forest reduction allows one tuple to generate multiple XML elements. To determine the effect of view-forest reduction on execution time, we generated 512 plans for $Q1_P$ and $Q2_P$ and then applied the view-forest reduction algorithm to each plan. Figures 27(b) and 28(b) contain the query-only times of the plans with view-forest reduction. These graphs should be compared to Figures 27(a) and 28(a), respectively. Note that view-forest reduction significantly

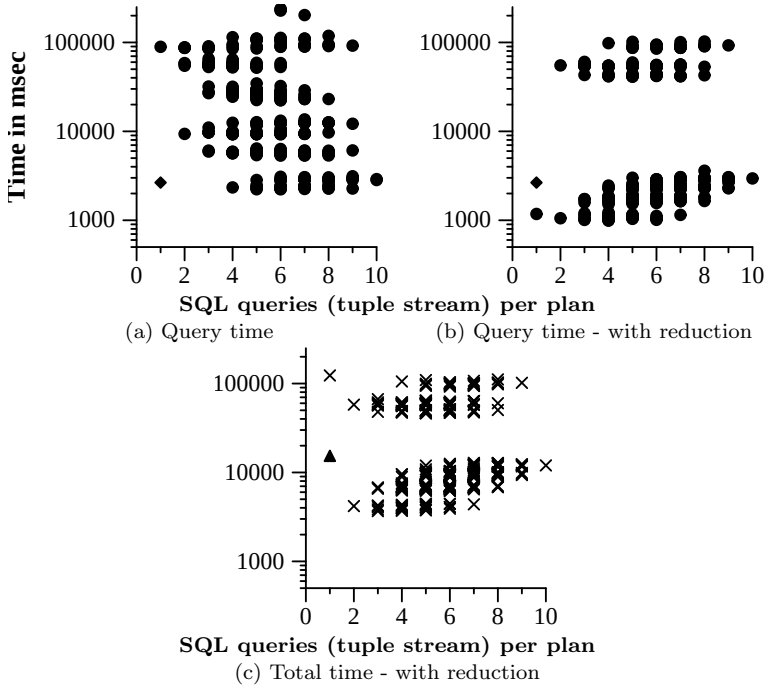


Fig. 27. $Q1_P$, Configuration A (times in msec)

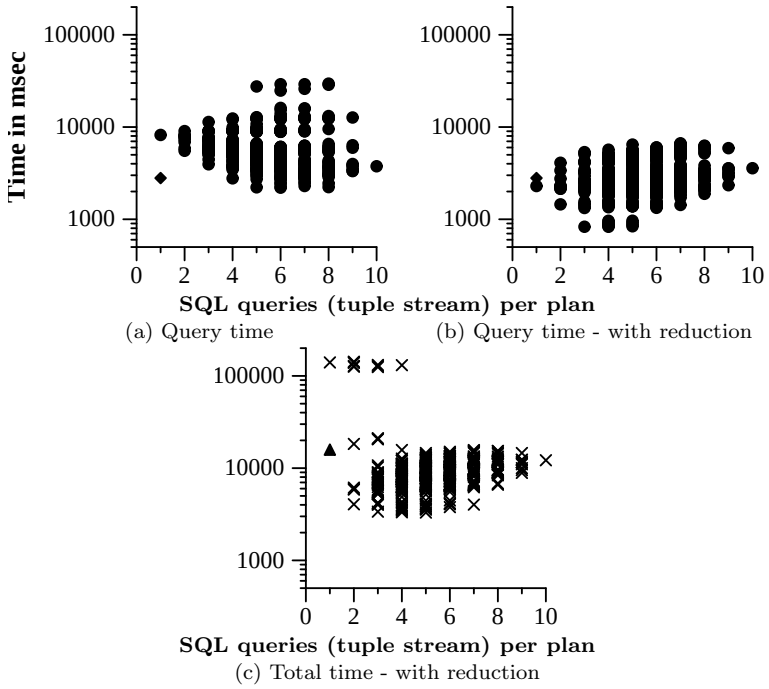
reduces query-only time. For both $Q1_P$ and $Q2_P$, the ten fastest reduced plans are 2.5 times faster than the ten fastest non-reduced plans, and the optimal plans are 2.6 to 4.3 times faster than the outer-union and fully partitioned plans.

The differences for total execution times, which include data-transfer time, are similar. For $Q1_P$ in Figure 27(c), the unified outer-union (triangle) is four times slower than optimal, and the fully partitioned plan is three times slower. For $Q2_P$ in Figure 28(c), the unified outer-union plan is 4.8 slower than optimal, and the fully partitioned plan is 3.7 times slower.

We note that for query-only time, the unified outer-union plan is only slightly slower than the unified outer-join plan, but its total execution time is much faster. The outer-join plan actually produces fewer, but wider, tuples than the outer-union plan; the additional width may induce anomalous caching behavior in JDBC. This suggests that we could further improve the total running time of the best plans if we rewrite them from outer joins to outer unions.

5.5 Plan-Generation Algorithm

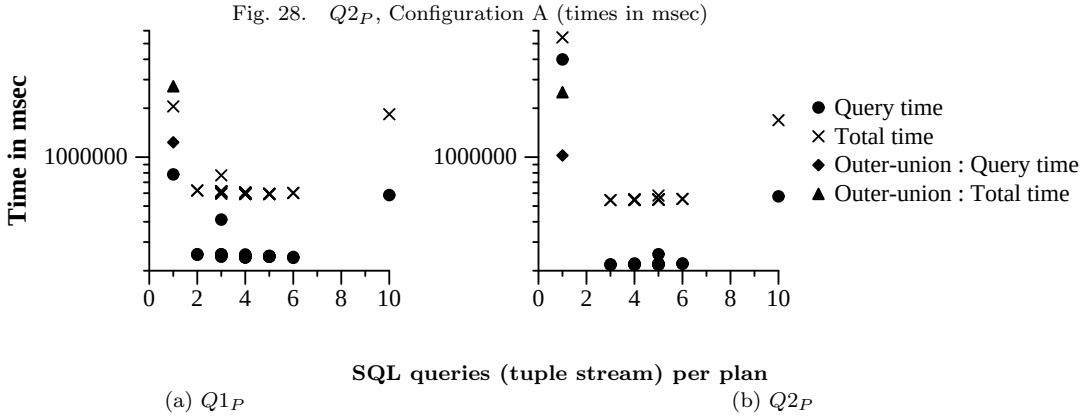
The experiments indicate that choosing a default unified, fully partitioned, or purely heuristic execution plan is not effective in practice and that devising an algorithm to generate near-optimal plans is worthwhile. The graphs in Figure 27 and 28 also suggest that there are many near-optimal queries. The only reliable source of query costs is the relational engine. If the relational engine can estimate the cost of a query, we can use the engine to choose "good" edges in a view forest, i.e., those



(a) Query time (b) Query time - with reduction

(c) Total time - with reduction

Fig. 28. $Q2_P$, Configuration A (times in msec)



(a) $Q1_P$ (b) $Q2_P$

Fig. 29. Configuration B, with view-forest reduction (times in msec)

edges whose two associated queries are less expensive to evaluate together than separately.

We present an algorithm that given a view tree, returns an evaluation plan that contains a set of mandatory view-tree edges and a set of optional view-tree edges. The algorithm uses the relational engine to estimate the relative cost of an edge in the view tree. For an edge $e = (nid_1, nid_2)$, where nid_1, nid_2 are the node identifier associated with the edge's parent and child nodes respectively, we compare the sum of the costs of evaluating the queries associated with nid_1 and nid_2 to the cost of

Edge = NID × NID	View-tree edge is pair of node identifiers
Node = NID × SQL	Node is identifier and complete SQL query
Tree = Edge × Node	View tree contains set of edges and nodes
Cost = Int × Edge × SQL	Cost of an edge, the edge, and the query if the edge is collapsed
`getQuery` : NID × SQL → SQL	Returns the complete SQL query
`incidentEdge` : E : Edge × e : Edge → [Edge]	Returns edges in E incident to e
`combineQueries` : SQL × SQL × Edge → SQL	Combines two queries on given edge into one query
`addEdge` : E : Edge × *plan* : Edge	Sorts edges in E by costs and adds qualifying edge to *plan*
`genPlan` : Tree × Float × Float × Float × Float → Edge × Edge	Returns plan containing mandatory and optional edges

Fig. 30. Types and functions of greedy algorithm

evaluating the two queries combined. We use a simple linear equation to estimate a query's cost:

$$\text{cost}(q, a, b) = a * \text{evaluation\_cost}(q) + b * \text{data\_size}(q)$$

$$\text{data\_size} = f(|\text{attrs}(q)| * \text{cardinality}(q))$$

The coefficients a and b assign weights to the query evaluation cost and query data size, respectively. The relational engine serves as an oracle, providing the values for the functions `evaluation_cost` and `cardinality`. This technique is feasible, because most commercial databases provide support for estimating these costs.

Figure 31 contains the plan-generation algorithm `genPlan`. Figure 30 contains the type signatures for the algorithm's functions. The function `genPlan` takes a view tree *ViewTree*, the cost coefficients a and b described above, and two thresholds: t_1 is the maximum threshold for a mandatory edge and t_2 is the maximum threshold for an optional edge. The recursive function `addEdge` takes the current set of edges (*Edges*), the query fragments associated with those edges (*Queries*), and the current sets of mandatory and optional edges. On each recursive invocation, `addEdge` computes the relative costs of every edge e_i in *Edges*:

$$\textit{cost} = \texttt{cost}(q_c) - (\texttt{cost}(q_1) + \texttt{cost}(q_2))$$

where q_1 and q_2 are the queries associated with e_i's parent and child nodes, and q_c is the result of combining q_1 and q_2. These costs are then sorted and `addPlan` considers the edge e with smallest relative cost (i.e., the one with greatest combined benefit). If the relative cost of e is less than t_1, the maximum threshold of a mandatory edge, then e is added greedily to the mandatory edges of the plan. Similarly, if e's relative cost is less than t_2, it is added to the optional edges of the plan. The function `addEdge` greedily adds edges until no remaining edge is less than the mandatory or optional threshold.

The function `combineQueries` determines how to collapse two queries into one query based on the label of the edge in the view tree. The `1`-labeled edges correspond to inner joins and the `*`-labeled edges to outer joins. In addition, `combineQueries` applies view-forest reduction to eligible edges.

```
function genPlan(ViewTree, t1, t2, a, b) {
  function addEdge(Edges, Queries, mandE, optE) {
    // Compute relative cost of each edge in Edges
    costE :  {Cost} = ⋃
      for e_i in Edges {
        let (nid1, nid2) = e_i
                q1 = getQuery(nid1, Queries)
                q2 = getQuery(nid2, Queries)
                qc = combineQueries(q1, q2, e_i)
        in (cost(qc) - (cost(q1) + cost(q2)), e_i, qc)
      }
    // Sort edges by costs
    sortedE = sort costE
    // Greedily add "best" edge to plan
    (i, e, qc) = head(sortedE)
    if (i < t1 || i < t2) {
      let (nid_q, body_q) = qc
        // Add e to plan
        mandE' = if (i < t1) mandE ∪ {e} else mandE
        optE' = if (i >= t1 && i < t2) optE ∪ {e}
                else optE
        (nid1, nid2) = e
        // Remove edge e from Edges
        Edges' = Edges - {e}
        // Remove e's queries from Queries
        Queries' = (Queries - { getQuery(nid1, Queries) }) -
                { getQuery(nid2, Queries) }
        // Add combined query qc to Queries
        Queries'' = Queries' ∪ {qc}
        // Remove edges incident to e from Edges
        incidentE = incidentEdge(Edges, e)
        Edges'' = Edges' - incidentE
        // For each edge incident to e, add new edge
        // that is incident to combined node defined
        // by query qc
        Edges''' = Edges'' ⋃
                for i in incidentE {
                    let (nid_u, nid_v) = i in
                    if (nid_u == nid1 || nid_u == nid2)
                    { (nid_q, nid_v) }
                    else { (nid_u, nid_q) }
                }

      in addEdge(Edges''', Queries'', mandE', optE')
    } else (mandE, optE)
  }
  let (Edges, Queries) = ViewTree
  in addEdge (Edges, Queries, {}, {})
}
```

Fig. 31. Greedy algorithm for plan generation

The complexity of the function `genPlan` is $O(|Edges|^2)$, because `addEdge` recomputes the costs of every edge in the view tree on each recursive call. To simplify presentation, this algorithm recomputes all the edge costs on each invocation, but the algorithm used in SilkRoute recomputes only the costs of those edges incident to each edge e selected by `addEdge`, which makes the complexity $O(n|Edges|)$ for n-ary trees. We expect that, in practice, the algorithm requires fewer computations, because it halts when no edge is worth collapsing.

5.6 Results

We applied the plan-generation algorithm twice to the view trees for $Q1_P$ and $Q2_P$: in one case, `combineQueries` did not apply view-forest reduction and in the second, it did. The generated plans for $Q1_P$ appear in Figure 32 (a) and (b), and in Figure 32 (c) and (d) for $Q2_P$[6].

The most important result is that the generated plans correspond directly to the fastest plans measured in Section 5.4. For $Q1_P$, the plans generated from the non-reduced and reduced view trees correspond to the fastest 32 plans. For $Q2_P$, the plans generated from the non-reduced view tree correspond to the fastest 32 plans, and the plans generated from the reduced view tree correspond to the first 31 and the 34th fastest plans. In Configuration B, the size of the database was 100 MB, so it was not possible to exhaustively test all 512 plans. Instead, we ran the greedy algorithm using view-forest reduction and compared the generated plans with the unified and fully partitioned plans. Sixteen plans were generated for $Q1_P$; they appear in Figure 32 (b). (Each subset of the four optional edges defines a plan.) Eight plans were generated for $Q2_P$; they appear in Figure 32 (d). Figure 29 plots the query-only and total-execution times for these plans and for the unified outer-union and fully partitioned plans.

For $Q1_P$ in Figure 29(a), the query-only time of the outer-union was five times slower than the optimal plan and the fully partitioned plan 2.4 times slower. For $Q2_P$, the differences were similar; the outer-union plan was 4.7 times slower than the optimal plan and the fully partitioned plan was 2.6 times slower. These results indicate that as the size of the XML view increases, generating optimal plans becomes imperative. Comparing total execution times of $Q1_P$ and $Q2_P$, the outer-union plan was 4.6 times slower and the fully partitioned plan 3.1 times slower.

For all the plans generated, we used the same values for the coefficients a (100) and b (1) and the thresholds t_1 (-60000) and t_2 (6000), which indicates that the linear cost function depends primarily on the characteristics of the database environment, and not on the characteristics of the query. Further experiments using a larger set of test queries are necessary to confirm this hypothesis.

Recall that the complexity of the plan-generation algorithm is $O(|Edges|^2)$ and that on *each* edge access, the algorithm requests the *estimated* costs of evaluation time and data size from the target database's query optimizer. For Queries 1 and 2, we found that the actual number of database requests for query-cost estimates were much smaller than the expected number of requests ($9^2 = 81$). Both Queries 1 and 2 required 22 requests for the non-reduced view tree and 25 requests for the reduced view tree.

[6]The original view trees are given in Figure 36

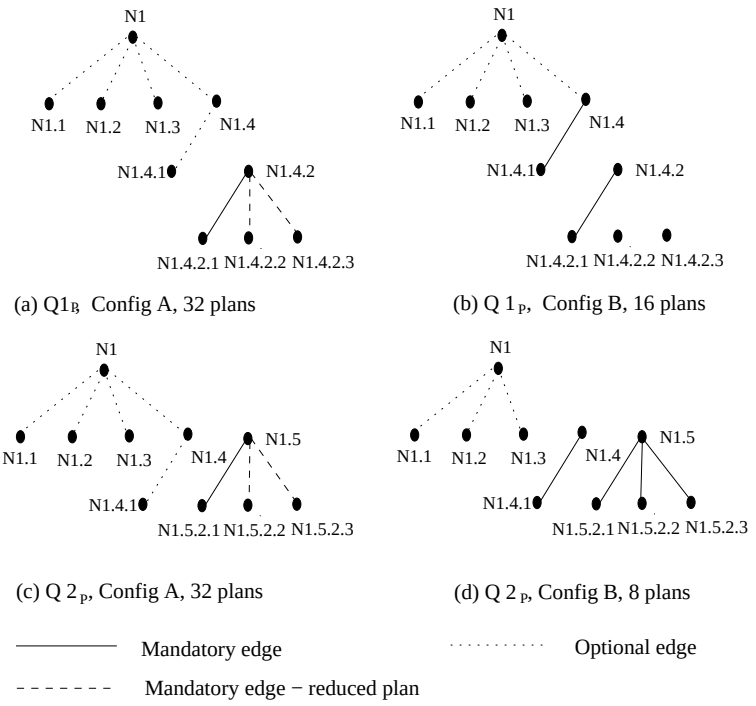


Fig. 32. Plans selected by Greedy Algorithm

6. ALTERNATIVE TECHNIQUES AND DISCUSSION

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.

The most widely used Web interfaces to relational databases are HTML forms with CGI scripts. User inputs are translated by a script into SQL queries, and their answers are rendered in HTML. The answers could be generated just as easily in XML. Forms interfaces are appropriate for casual users, but inappropriate for data exchange between applications, because they limit the application to only those queries provided 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. The ROLEX system[Bohannon et al. 2002] provides a dynamic, virtual DOM interface to relational data. ROLEX's view abstraction (the "schema-tree query") is based on our view forest, but their query evaluation strategies and indices are customized for navigational access. This technique is especially effective for interactive Web services in which small fragments of the XML view are materialized.

A third alternative is to use a native XML database engine, which can warehouse XML data and process XQuery queries directly in the native XML engine. XML engines will not replace relational databases, but a high-performance XML engine

might be appropriate to use in data exchange. For example, one could materialize an XML view using SilkRoute and store the result in an XML engine that supports XQuery, thus avoiding the composition cost done in SilkRoute. We do not expect, however, XML engines to match the performance of commercial SQL engines anytime soon. In addition, this data-warehouse approach can suffer from data staleness and incurs a high space overhead because it duplicates all the data in XML.

Three commercial XML publishing systems, Oracle XML SQL Utility [Wait 1999], IBM DB2 XML Extender [Extender 2000], and Microsoft SQL Server 2000 [Rys 2000], support features similar to those provided by SilkRoute. Oracle's XSQL embeds individual SQL queries in XSLT [World-Wide Web Consortium 1999] stylesheets. The result of the SQL query is emitted in a canonical XML format, and the stylesheet converts the XML into the desired view. Of these systems, XSQL couples most tightly the XML view to the corresponding SQL queries. Although not general, this solution is efficient, because the relational engine evaluates the entire query and constructs the canonical XML in-engine. The IBM DB2 Data Access Definition (DAD) language, has a data extraction part and an XML template. Each element in the XML template may contain arbitrary selection and join conditions on the relational tables, but the criteria for grouping elements is implicit in the DAD, and DAD specifications cannot be nested arbitrarily, which makes it considerably less expressive than XQuery. SQL Server 2000 supports XML view mechanisms like the two described above. In addition, the user may construct the unified SQL plan by hand. This effectively hard wires the evaluation plan into the view, but it allows the user to define arbitrarily complex XML views. Hand-written unified queries are similar to those constructed automatically by SilkRoute's plan-generation algorithm. SQL Server's XML views are selective, because SQL Server permits querying of the XML view using XPath. As a user-query language, XPath supports selection and projection of elements, but not restructuring or grouping as does XQuery. Although no commercial system currently supports the full generality of XQuery, these three vendors are working actively to provide an XQuery-interface for application queries.

XQuery can express the transformations provided by the three XML publishing tools described above, and therefore, SilkRoute's view forest representation of XML view queries captures the XML mappings in all these systems. Our greedy optimization algorithm takes a view forest as input, and therefore could be directly applied to the XML view definitions expressed by these tools.

From a theoretical perspective, query composition is simple for select-project-join queries [Abiteboul et al. 1995; Ramakrishnan and Gehrke 2000], and for the relational calculus [Abiteboul et al. 1995]. In the context of semistructured data, Papakonstantinou et al. first address the problem in the framework of MSL [Papakonstantinou et al. 1996], 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. [Deutsch et al. 1999] and Papakonstantinou and Vassalos [Papakonstantinou and Vassalos 1999] address query composition in the more complex setting of query rewriting for semistructured data. Our solution borrows ideas from [Deutsch et al. 1999; Papakonstantinou et al. 1996].

The research system XPERANTO [Carey et al. 2000] is very similar in applicability and spirit to SilkRoute. XPERANTO supports definition of XML views of relational data using XQuery and supports composition of application queries over an XML view [Shanmugasundaram et al. 2001]. XPERANTO uses an "XML Query Graph Model" (XQGM) as its intermediate representation of a view. The XQGM is analogous to a physical execution plan produced by a query optimizer. Nodes in the XQGM represent operations in an algebra (e.g., select, join, unnest, union) and edges represent the data flow from one operation to the next. Individual operations may invoke "XML-aware" procedures for constructing and deconstructing XML values. Whereas SilkRoute's view forest is entirely declarative, the XQGM is more imperative and procedural, which may make it less amenable to composition with an arbitrary number of XQuery queries. That is, it may not be possible to always produce an XQGM that may be composed with another XQuery query. The XQGM, however, may better capture the relationship between XQuery expressions and complex SQL expressions than does a view forest. The two representations may be symbiotic: declarative view forests are appropriate for the "front end" query composition whereas the procedural XQGM may be better for "back end" SQL generation.

XPERANTO has also addressed efficient evaluation of XML views. Shanmugasudaram et al. [Shanmugasundaram et al. 2000] describe several methods for computing XML views with relational engines. They classify the methods along three axis: early/late structuring, early/late tagging, and in-engine/outside-engine XML generation. They consider a variety of algorithms and compare them experimentally. In the unordered outer union strategy, the XML generator uses a main memory hash table to assemble the XML objects, which requires the XML view fit in main memory. In CLOB de-correlated queries, the XML result is constructed by the relational engine, which is also effective when the XML view fits in main memory. The best overall performance is achieved by the CLOB de-correlated algorithm, the unsorted outer union, and the sorted outer union. Of these, only the sorted outer union applies to large XML views that exceed main memory.

Commercial database vendors and database researchers have intense interest in the problems of publishing, storing, querying, and warehousing XML data. Although XML is still an immature technology, there is real demand for high-performance XML storage and query engines. This demand will only increase as XML becomes an integral part of business-to-business applications. The main contributions of SilkRoute have been to characterize the fundamental problems of viewing and querying XML data stored in relational database systems; to present a framework for publishing relational data in XML; to create an abstraction, the view forest, that captures the semantics of an XQuery query and that supports query composition and evaluation; and to develop algorithms for composing and evaluating XML views.

A. QUERIES FOR EXPERIMENTS

The TPC Benchmark 'H' database [Council 2001] contains information about parts, the suppliers of those parts, customers, and their part orders. Figure 33 contains a fragment of the database's schema.

```
Supplier(suppkey CHAR(10) PRIMARY KEY, name VARCHAR(20), addr VARCHAR(50),
         nationkey CHAR(10))
PartSupp(partkey CHAR(10) PRIMARY KEY, suppkey PRIMARY KEY CHAR(10),
         availqty INTEGER)
Part(partkey CHAR(10) PRIMARY KEY, name VARCHAR(20), mfgr VARCHAR(20),
     brand VARCHAR(20), size VARCHAR(20), retail REAL)
Customer(custkey CHAR(10) PRIMARY KEY, name VARCHAR(20), addr VARCHAR(50),
         nationkey CHAR(10), phone CHAR(10))
LineItem(orderkey CHAR(10) PRIMARY KEY, partkey CHAR(10),
suppkey CHAR(10), ltemno CHAR(10), qty INTEGER, price REAL)
Orders(orderkey CHAR(10) PRIMARY KEY, custkey CHAR(10), status CHAR(10),
       price REAL, date DATE)
Nation(nationkey CHAR(10) PRIMARY KEY, name VARCHAR(20), regionkey CHAR(10))
Region(regionkey CHAR(10) PRIMARY KEY, name VARCHAR(20))
```

Fig. 33. Fragment of TPC-H Schema

```
element suppliers {
  element supplier*
}
element supplier {
  element name,
  element nation,
  element region,
  element part*
}
element name     { string }
element nation   { string }
element region   { string }
```

```
element part     {
  element name,
  element order*
}
element order    {
  element orderkey,
  element customer,
  element cnation
}
element orderkey { string }
element customer { string }
element cnation  { string }
```

Fig. 34. Schema in XQuery type notation of public view $Q1_P$ in Figure 35

We first explain the public query $Q1_P$. Figure 34 gives the XML schema for this view. Each `supplier` element includes its name, its `nation`, the geographical `region` of the nation, and a list of the supplier's `parts`. Each `part` element includes a part name and a list of orders pending for the part. Each `order` element includes an orderkey, the associated customer, and the customer's nation. The `name`, `nation`, `region`, and `customer` elements all contain strings. Figure 35 contains the public query corresponding to the schema in Figure 33.

Figure 36 (top) depicts the view tree for the $Q1_P$ in Figure 35. In this view tree, there are nine edges and 2^9 or 512 subsets of edges, each of which corresponds to a partition of the tree. Therefore there are 512 possible plans for splitting the tree into a collection of SQL queries; each plan consists of between 1 and 10 tuple streams.

$Q2_P$ is identical to $Q1_P$ except that the block defining the `order` node is a child of the `supplier` node instead of the `part` node. Figure 36 (bottom) depicts its view tree. In $Q1_P$, the two one-to-many edges (labeled "*"), are nested in a chain, whereas in $Q2_P$, the two '*' edges are parallel.

```
for $s in $CanonicalView/Supplier/Tuple
return
  <supplier> <name> { $s/name } </name>
    {for $n in $CanonicalView/Nation/Tuple
      where $s/nationkey = $n/nationkey
      return
        <nation>{ data($n/name) } </nation>
        {for $r in $CanonicalView/Region/Tuple
          where $n/regionkey = $r/regionkey
          return <region>{ data($r.name) }</region> }  }
    {for $ps in $CanonicalView/PartSupp/Tuple, $p in $CanonicalView/Part/Tuple
      where $s/suppkey = $ps.suppkey,
            $ps/partkey = p/partkey
      return
        <part> <name> { data($p.name) }</name>
          {for $l in $CanonicalView/LineItem/Tuple, $o in $CanonicalView/Orders/Tuple
            where $ps/partkey = $l/partkey,
                  $ps/suppkey = $l/suppkey,
                  $l/orderkey = $o/orderkey
            return
              <order>
                <orderkey>{ data($o/orderkey) }</orderkey>
                {for $c in $CanonicalView/Customer/Tuple
                  where $o/custkey = $c/custkey
                  return
                    <customer>{ data($c/name) }</customer>
                    {for $n2 in $CanonicalView/Nation/Tuple
                      where $c/nationkey = $n2/nationkey
                      return
                      <cnation>{ data($n2/name) }</cnation>
                    }
              }
              </order> }
        </part> }
  </supplier>
```

Fig. 35. Public query corresponding to schema in Figure 33 ($Q1_P$)

ACKNOWLEDGMENTS

We thank Jayavel Shanmugasundaram for his insightful remarks on the conference papers that contributed to this paper, and we thank the reviewers for their detailed comments and suggestions.

REFERENCES

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

Beeri, C. and Bernstein, P. 1979. Computational problems related to the design of normal form relational schemes. *ACM Transactions on Database Systems 4,* 1, 30–59.

Bohannon, P., Ganguly, S., Korth, H., et al. 2002. Optimizing view queries in ROLEX to support navigable result trees. In *Proceedings of Very Large Data Bases*. VLDB, Hong Kong.

Carey, M. et al. 2000. Xperanto: Middleware for publishing object-relational data as XML documents. In *Proceedings of Very Large Data Bases*. VLDB, Cairo, Egypt, 646–648.

Choi, B., Fernandez, M., and Simeon, J. 2002. The xquery formal semantics:

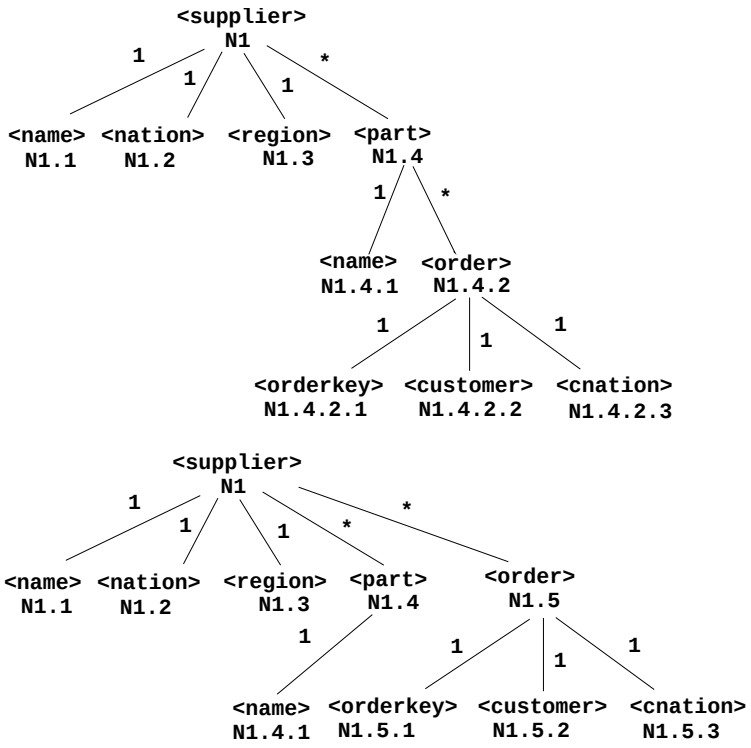


Fig. 36. Labeled view tree for $Q1_P$ (top) and $Q2_p$ (bottom)

a foundation for implementation and optimization. Tech. rep., AT&T Labs Research. Submitted for publication.

Council, T. P. P. 2001. TPC-H (ad-hoc, decision support) benchmark. http://www.tpc.org/.

Deutsch, A., Fernández, M., and Suciu, D. 1999. Storing semistructured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, Amsterdam, the Netherlands, 431–442.

Extender, I. D. U. D. X. 2000. XML extender administration and programming. (http://www-4.ibm.com/software/data/db2/extenders/xmlext/docs/v71wrk/english/index.htm).

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML*. MIT Press.

Online Library. *Introduction to the Dewey Decimal Classification*. Online Computer Library Center. http://www.oclc.org/oclc/fp/about/about_the_ddc.htm.

Papakonstantinou, Y., Abiteboul, S., , and Garcia-Molina, H. 1996. Object fusion in mediator systems. In *Proceedings of Very Large Data Bases*. VLDB, Bombay, India, 413–424.

Papakonstantinou, Y. and Vassalos, V. 1999. Query rewriting for semistructured data. In *Proceedings ACM SIGMOD International Conference on Management of Data*. ACM, Philadelphia, PA, 455–466.

Pirahesh, H., Leung, T. Y., and Hasan, W. 1997. A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. In *Proceedings of the Thirteenth International Conference on Data Engineering*. Birmingham, UK, 391–400.

Ramakrishnan, R. and Gehrke, J. 2000. *Database Management Systems*, 2nd ed. McGraw Hill. Section 14.5, page 401.

Rys, M. 2000. Support webcast: Microsoft sql server 2000: New XML features. (http://support.microsoft.com/servicedesks/ Webcasts/wc042800/wcblurb042800.asp).

SAHUGUET, A. 2000. Everything you ever wanted to know about dtds, but were afraid to ask. In *International Workshop on the Web and Databases (WebDB'2000)*.

SHANMUGASUNDARAM, J., KIERNAN, J., SHEKITA, E., ET AL. 2001. Querying XML views of relational data. In *Proceedings of Very Large Data Bases*. VLDB, Roma, Italy, 261–270.

SHANMUGASUNDARAM, J., SHEKITA, E., BARR, R., PIRAHESH, H., AND REINWALD, B. 2000. Efficiently publishing relational data as XML documents. In *Proceedings of Very Large Data Bases*. VLDB, Cairo, Egypt, 65–76.

WAIT, B. 1999. Using XML in oracle database applications. (`http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about_xml.htm`) Oracle Corporation.

World-Wide Web Consortium 1999. *XSL Transformations (XSLT), Version 1.0, W3C Recommendation*. World-Wide Web Consortium. `http://www.w3.org/TR/xslt`.

World-Wide Web Consortium 2001a. *XML Schema Part 1: Structures, W3C Recommendation*. World-Wide Web Consortium. `http://www.w3.org/TR/xmlschema-1`.

World-Wide Web Consortium 2001b. *XML Schema Part 2: Datatypes, W3C Recommendation*. World-Wide Web Consortium. `http://www.w3.org/TR/xmlschema-2`.

World-Wide Web Consortium 2002a. *XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft*. World-Wide Web Consortium. `http://www.w3.org/TR/query-datamodel/`.

World-Wide Web Consortium 2002b. *XQuery 1.0 and XPath 2.0 Functions and Operators Version 1.0, W3C Working Draft*. World-Wide Web Consortium. `http://www.w3.org/TR/xquery-operators/`.

World-Wide Web Consortium 2002c. *XQuery 1.0 Formal Semantics, W3C Working Draft*. World-Wide Web Consortium. `http://www.w3.org/TR/query-semantics/`.

World-Wide Web Consortium 2002d. *XPath 2.0, W3C Working Draft*. `http://www.w3.org/TR/xpath20/`.

World-Wide Web Consortium 2002e. *XQuery 1.0: An XML Query Language, W3C Working Draft*. `http://www.w3.org/TR/xquery/`.

Received December 2001; revised June 2002; accepted August 2002.