Quilt: An XML Query Language for Heterogeneous Data Sources

Don Chamberlin[1], Jonathan Robie[2], and Daniela Florescu[3]

[1] IBM Almaden Research Center, San Jose, CA 95120, USA
chamberlin@almaden.ibm.com
[2] Software AG – USA, 3207 Gibson Road, Durham, NC 27703
Jonathan.Robie@SoftwareAG-USA.com
[3] INRIA, 78153 Le Chesnay cedex, France
Daniela.Florescu@inria.fr

Abstract. The World Wide Web promises to transform human society by making virtually all types of information instantly available everywhere. Two prerequisites for this promise to be realized are a universal markup language and a universal query language. The power and flexibility of XML make it the leading candidate for a universal markup language. XML provides a way to label information from diverse data sources including structured and semi-structured documents, relational databases, and object repositories. Several XML-based query languages have been proposed, each oriented toward a specific category of information. Quilt is a new proposal that attempts to unify concepts from several of these query languages, resulting in a new language that exploits the full versatility of XML. The name Quilt suggests both the way in which features from several languages were assembled to make a new query language, and the way in which Quilt queries can combine information from diverse data sources into a query result with a new structure of its own.

1 Introduction

The Extensible Markup Language, XML[1], is having a profoundly unifying effect on diverse forms of information. For the first time, XML provides an information interchange format that is editable, easily parsed, and capable of representing nearly any kind of structured or semi-structured information.

As an example of the unifying influence of XML, consider the once-divergent worlds of documents and databases. Documents have irregular structure, are deeply nested, use relatively simple datatypes, and place great importance on ordering. Relational databases, on the other hand, have a very regular structure, are relatively flat, use complex datatypes, and usually place little importance on ordering. It is a tribute to the flexibility of XML that it is pulling together these diverse forms of information to the extent that the distinction between a document and a database is quickly vanishing.

In order to realize its potential as a universal format for information exchange, XML needs a query language that is as flexible as XML itself. For querying documents, the language needs to be able to preserve order and hierarchy. For querying databases, the language needs to provide traditional database operations such as joins and grouping. The language must be capable of dealing with all the information structures found in the XML Schema specification[2], and must able to transform information from one structure into another.

Our goal is to design a small, implementable language that meets the requirements identified by the W3C XML Query Working Group[3]. We want a language in which queries are concise but readable. We also want a language that is flexible enough to query a broad spectrum of XML information sources, and we have used examples from the database and document communities as representative of these requirements.

Our strategy in designing the language has been to borrow features from several other languages that seem to have strengths in specific areas. From XPath[4] and XQL[5] we take a syntax for navigating in hierarchical documents. From XML-QL[6] we take the notion of binding variables and then using the bound variables to create new structures. From SQL[7] we take the idea of a series of clauses based on keywords that provide a pattern for restructuring data (the SELECT-FROM-WHERE pattern in SQL). From OQL[8] we take the notion of a functional language composed of several different kinds of expressions that can be nested with full generality. We have also been influenced by reading about other XML query languages such as Lorel[9] and YATL[10]. We decided to name our language Quilt because of its heritage as a patchwork of features from other languages, and also because of its goal of assembling information from multiple diverse sources. Quilt has also been described in [11].

The W3C XML Query Working Group has identified a requirement for both a human-readable query syntax and an XML-based query syntax. Quilt is designed to meet the first of these requirements. We recognize that an alternative, XML-based syntax for the Quilt semantics would be useful for some applications.

2 The Quilt Language

Like OQL, Quilt is a functional language in which a query is represented as an expression. Quilt supports several kinds of expression, and therefore a Quilt query may take several different forms. The various forms of Quilt expressions can be nested with full generality, so the notion of a "subquery" is natural to Quilt.

The input and output of a Quilt query are XML documents, fragments of XML documents, or collections of XML documents. We can think of these inputs and outputs as instances of a data model called the XML Query Data Model, which is under development by the W3C XML Query Working Group[12]. This data model is a refinement of the data model described in the XPath specification[4], in which a document is modeled as a tree of nodes. A fragment of a document, or a collection of documents, may lack a common root and may therefore be modeled as an ordered forest of nodes of various types, including element nodes, attribute nodes, and text nodes, as illustrated in Figure 1.

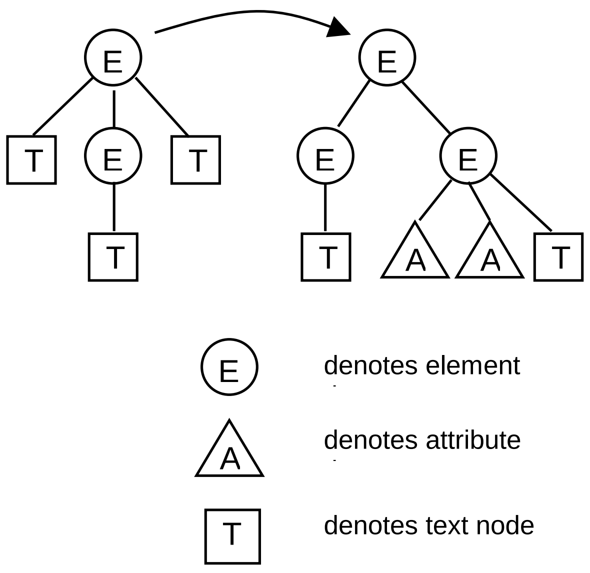

Fig. 1. An instance of the XML Query Data Model: an ordered forest

An informal syntax for Quilt is given in the Appendix of this paper. Formal definition of the syntax and semantics of the language should be considered a work in progress.

The principal forms of Quilt expressions are as follows:

1. Path expressions
2. Element constructors
3. FLWR expressions
4. Expressions involving operators and functions
5. Conditional expressions
6. Quantifiers
7. Variable bindings

Each of these types of expressions is introduced and explained by a series of examples in the following sections. For syntactic details, please refer to the Appendix.

A Quilt query may contain a comment, which is ignored during query processing. As in SQL, the beginning delimiter of a comment is a double hyphen and the ending delimiter is a newline character.

2.1 Path Expressions

Quilt path expressions are based on the abbreviated syntax of XPath, which provides a way to navigate through a hierarchy of nodes. The result of a path expression is an ordered forest consisting of those nodes that satisfy the expression and their descendants. XPath syntax is used in several XML-related applications such as XSLT [13] and XPointer [14].

As in XPath, a Quilt path expression consists of a series of *steps.* Each step represents movement through a document in a particular direction, and each step can apply a predicate to eliminate nodes that fail to satisfy a given condition. The result of each step is a set of nodes that serves as a starting point for the next step.

A path expression can begin with an expression that identifies a specific node, such as the function document(string), which returns the root node of a named document. A Quilt query can also contain a path expression beginning with "/" or "//" which represents an implicit root node, determined by the environment in which the query is executed.

A complete discussion of Xpath abbreviated syntax can be found in [4]. Briefly, the following symbols are used:

.	Denotes the current node.
..	Denotes the parent of the current node.
/	Denotes the root node, or children of the current node
//	Denotes descendants of the current node (closure of /).
@	Denotes attributes of the current node
*	Denotes "any" (node with unrestricted name)
[ ]	Brackets enclose a Boolean expression that serves as a predicate for a given step.
[n]	When a predicate consists of an integer, it serves to select the element with the given ordinal number from a list of elements.

The following example uses a path expression consisting of three steps. The first step locates the root node of a document. The second step locates the second chapter element that is a child of the root element. The third step finds figure elements occurring anywhere within the chapter, but retains only those figure elements that have a caption with the value "Tree Frogs."

(Q1) In the second chapter of the document named "zoo.xml", find the figure(s) with caption "Tree Frogs".

```
document("zoo.xml")/chapter[2]
   //figure[caption = "Tree Frogs"]
```

It is sometimes desirable to return a set of elements whose ordinal numbers span a certain range. For this purpose, Quilt provides a RANGE predicate that is adapted from one of the features of XQL[5], as illustrated in the following example:

(Q2) Find all the figures in chapters 2 through 5 of the document named "zoo.xml."

```
document("zoo.xml")/chapter[RANGE 2 TO 5]//figure
```

In addition to the operators of the XPath abbreviated syntax, Quilt introduces an operator called the dereference operator ("->"). When a dereference operator follows an IDREF-type attribute or a key, it returns the element(s) that are referenced by the attribute or key. Dereference operators can be used in the steps of a path expression.

For example, the following query uses a dereference operator to find the element referenced by the "refid" attribute of a "figref" element.

(Q3) Find captions of figures that are referenced by <figref> elements in the chapter of "zoo.xml" with title "Frogs".

```
document("zoo.xml")/chapter[title = "Frogs"]
   //figref/@refid->/caption
```

The Quilt dereference operator is similar in purpose to the id function of XPath. However, the right-arrow notation is intended to be easier to read, especially in path expressions that involve multiple dereferences. For example, suppose that a given document contains a set of <emp> elements, each of which may contain a "mentor" attribute. The "mentor" attribute is of type IDREF, and it references another <emp> element that represents the mentor of the given employee. The name of each employee is represented by a <name> element nested inside the <emp> element.

(Q4) Find the name of the mentor of the mentor of the employee named "Jack".

```
/emp[name = "Jack"]/@mentor->/@mentor->/name
```

As in XPath, the identifiers used in Quilt expressions can be qualified by namespace prefixes[15]. Quilt provides a syntax for declaring the Universal Resource Identifier (URI) associated with each namespace prefix used in a query, as illustrated in the following example:

(Q5) In the document "zoo.xml", find <tiger> elements in the abc namespace that contain any subelement in the xyz namespace.

```
NAMESPACE abc = "www.abc.com/names"
NAMESPACE xyz = "www.xyz.com/names"
document("zoo.xml")//abc:tiger[xyz:*]
```

2.2 Element Constructors

An element constructor is used to generate an element node. Similar constructors exist for other types of nodes such as comments and processing instructions. An element constructor consists of a start tag and an end tag, enclosing an optional list of expressions that provide the content of the element. The start tag may also specify the values of one or more attributes. The name of the start tag may be specified either by a constant or a variable.

Although an element constructor is an expression in its own right, its typical use is nested inside another expression that binds one or more variables that are used in the element constructor. Both of the following examples are query fragments that refer to variables that are bound in some enclosing expression.

(Q6) Generate an <emp> element containing an "empid" attribute and nested <name> and <job> elements. The values of the attribute and nested elements are specified by variables that are bound in other parts of the query.

```
<emp empid = $id>
   <name> $n </name> ,
   <job> $j </job>
</emp>
```

In the following example, the name of the generated element is specified by a variable named $tagname. Note that, when a start-tag contains a variable name, the matching end-tag must contain the same variable name (see Section 2.8 for a more interesting version of this example.)

(Q7) Generate an element with a computed name, containing nested elements named <description> and <price>.

```
<$tagname>
   <description> $d </description> ,
   <price> $p </price>
</$tagname>
```

2.3 FLWR Expressions

A FLWR (pronounced "flower") expression is constructed from FOR, LET, WHERE, and RETURN clauses. As in an SQL query, these clauses must appear in a specific order. A FLWR expression is used whenever it is necessary to iterate over the elements of a collection.

A FLWR expression begins by binding values to one or more variables, and then uses these variables to construct a result (in general, an ordered forest of nodes). The overall flow of data in a FLWR expression is illustrated in Figure 2.

A FLWR expression begins with a FOR-clause that generates one or more bindings for one or more variables. Each variable introduced in the FOR-clause is associated with an expression (for example, a path expression). In general, each of these expressions returns a list of nodes. The result of the FOR-clause is a list of tuples, each of which contains a binding for each of the variables. The variables are bound to individual nodes returned by their respective expressions, in such a way that the binding-tuples represent the cross-product of the node-lists returned by all the expressions.

The initial FOR-clause in a FLWR expression can be followed by one or more LET-clauses and additional FOR-clauses, which provide bindings for additional variables. A LET-clause simply binds one or more variables to the result of one or more expressions. Unlike a FOR-clause, which iterates over node-lists to generate many bindings for each variable, a LET-clause generates only one binding for each variable. Bindings generated by a FOR-clause bind each variable to a single node (with its descendants), whereas a LET-clause may bind a variable to a forest of nodes.

A FLWR expression may contain several FOR and LET-clauses, and each of these clauses may contain references to variables bound in previous clauses. The result of the sequence of FOR and LET clauses is an ordered list of tuples of bound variables. The number of tuples generated by a FOR/LET sequence is the product of the cardinalities of the node-lists returned by the expressions in the FOR-clauses. The tuples generated by the FOR/LET sequence have an order that is determined by the order of their bound elements in the input document, with the first bound variable taking precedence, followed by the second bound variable, and so on. However, if some expression used in a FOR-clause is unordered (for example, because it contains a distinct function), the tuples generated by the FOR/LET sequence are unordered.

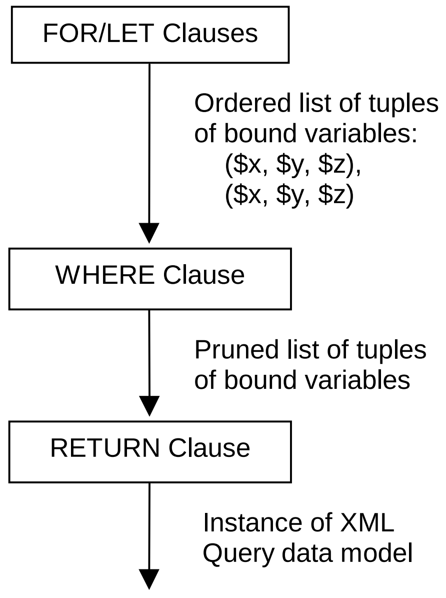
FOR/LET Clauses
Ordered list of tuples of bound variables: ($x, $y, $z), ($x, $y, $z)
WHERE Clause
Pruned list of tuples of bound variables
RETURN Clause
Instance of XML Query data model

Fig. 2. Flow of data in a FLWR expression

Each of the binding-tuples generated by the FOR and LET clauses is subject to further filtering by an optional WHERE-clause. Only those tuples for which the condition in the WHERE-clause is true are used to invoke the RETURN clause. The WHERE-clause may contain several predicates, connected by AND, OR, and NOT. These predicates usually contain references to the bound variables. Variables bound by a FOR-clause represent a single node (with its descendants) and so they are typically used in scalar predicates such as $p/color = "Red". Variables bound by a LET-clause, on the other hand, may represent collections of nodes, and can be used in collection-oriented predicates such as avg($p/price) > 100. The ordering of the binding-tuples generated by the FOR and LET clauses is preserved by the WHERE-clause.

The RETURN-clause generates the output of the FLWR expression, which may be a node, an ordered forest of nodes, or a primitive value. The RETURN-clause is executed once for each tuple of bindings that is generated by the FOR and LET-clauses and satisfies the condition in the WHERE-clause. If an ordering exists among these tuples, the RETURN-clause is executed on each tuple, in order, and the order of the results is preserved in the output document. The RETURN-clause contains an expression that often contains element constructors, references to bound variables, and nested subexpressions.

We will consider some examples of FLWR expressions based on a document named "bib.xml" that contains a list of <book> elements. Each <book> element, in turn, contains a <title> element, one or more <author> elements, a <publisher> element, a <year> element, and a <price> element. The first example is so simple that it could have been expressed using a path expression, but it is perhaps more readable when expressed as a FLWR expression.

(Q8) List the titles of books published by Morgan Kaufmann in 1998.

```
FOR $b IN document("bib.xml")//book
WHERE $b/publisher = "Morgan Kaufmann"
AND $b/year = "1998"
RETURN $b/title
```

Example Q9 uses a <u>distinct</u> function in the FOR-clause to eliminate duplicates from the list of publishers found in the input document. Two elements are considered to be duplicates if their values (including name, attributes, and normalized content) are equal. The result of the <u>distinct</u> function is an unordered set of elements. Example Q9 then uses a LET-clause to bind a variable to the average price of books published by each of the publishers bound in the FOR-clause.

(Q9) List each publisher and the average price of its books.

```
FOR $p IN distinct(document("bib.xml")//publisher)
LET $a := avg(document("bib.xml")
   /book[publisher = $p]/price)
RETURN
   <publisher>
      <name> $p/text() </name> ,
      <avgprice> $a </avgprice>
   </publisher>
```

The next example uses a LET-clause to bind a variable $b to a set of books, and then uses a WHERE-clause to apply a condition to the set, retaining only bindings in which $b contains more than 100 elements. This query also illustrates the common practice of enclosing a FLWR expression inside an element constructor which provides an enclosing element for the query result.

(Q10) List the publishers who have published more than 100 books.

```
<big_publishers>
   FOR $p IN distinct(document("bib.xml")//publisher)
   LET $b := document("bib.xml")/book[publisher = $p]
   WHERE count($b) > 100
   RETURN $p
</big_publishers>
```

FLWR expressions are often useful for performing structural transformations on documents, as illustrated by the next query, which inverts a hierarchy. This example also illustrates how one FLWR expression can be nested inside another.

(Q11) Invert the structure of the input document so that, instead of each book element containing a list of authors, each distinct author element contains a list of book-titles.

```
<author_list>
   FOR $a IN distinct(document("bib.xml")//author)
   RETURN
      <author>
         <name> $a/text() </name>,
         FOR $b IN document("bib.xml")//book
            [author = $a]
         RETURN $b/title
      </author>
</author_list>
```

By default, a Quilt query preserves the ordering of elements in the input document(s), as represented by the values of its bound variables. However, it is often important to specify an order for the elements in a query result that supplements or supercedes the order derived from the variable bindings. If a query result contains several levels of nested elements, an ordering may be required among the elements at each level. Quilt provides a SORTBY clause that may be used after an element constructor or path expression to specify an ordering among the resulting elements. The arguments of the SORTBY clause are evaluated within the context of the individual nodes to be sorted, and may be followed by ASCENDING or DESCENDING to specify the direction of the sort (ASCENDING is the default.) The use of SORTBY is illustrated by the following example.

(Q12) Make an alphabetic list of publishers. Within each publisher, make a list of books, each containing a title and a price, in descending order by price.

```
<publisher_list>
   FOR $p IN distinct(document("bib.xml")//publisher)
   RETURN
      <publisher>
         <name> $p/text() </name> ,
         FOR $b IN document("bib.xml")//book
            [publisher = $p]
         RETURN
            <book>
               $b/title ,
               $b/price
            </book> SORTBY(price DESCENDING)
      </publisher> SORTBY(name)
</publisher_list>
```

2.4 Operators in Expressions

Like most languages, Quilt allows expressions to be constructed using infix and prefix operators, and allows nested expressions inside parentheses to serve as operands. Quilt supports the usual set of arithmetic and logical operators, and the collection operators UNION, INTERSECT, and EXCEPT. The detailed semantics of these operators, as applied to various kinds of collections including sets, bags, and lists, is left to a more detailed language specification.

From XQL, Quilt inherits the infix operators BEFORE and AFTER, which are useful in searching for information by its ordinal position. Each instance of the XML Query data model (regardless of whether it is a complete document, a fragment of a document, or a list of documents) is a forest that includes a total ordering, called "document order," among all its nodes. BEFORE operates on two collections of elements and returns those elements in the first collection that occur before at least one element of the second collection in document order (of course, this is possible only if the two collections are subsets of the same data model instance.) AFTER is defined in a similar way. Since BEFORE and AFTER are based on global document ordering, they can compare the positions of elements that do not have the same parent. The next two examples illustrate the use of BEFORE and AFTER by retrieving excerpts from a surgical report that includes <procedure>, <incision>, and <anesthesia> elements.

(Q13) Prepare a "critical sequence" report consisting of all elements that occur between the first and second incision in the first procedure.

```
<critical_sequence>
   FOR $p IN //procedure[1],
       $e IN //* AFTER ($p//incision)[1]
          BEFORE ($p//incision)[2]
   RETURN shallow($e)
</critical_sequence>
```

The <u>shallow</u> function strips an element of its subelements.

(Q14) Find procedures in which no anesthesia occurs before the first incision.

```
-- Finds potential lawsuits
FOR $p in //procedure
WHERE empty($p//anesthesia BEFORE ($p//incision)[1])
RETURN $p
```

Another important operator introduced by Quilt is the FILTER operator. FILTER takes two operands, each of which is an expression that, in general, evaluates to an ordered forest of nodes. FILTER returns a subset of the nodes in the forest represented by the first operand, while preserving the hierarchic and sequential relationships among these nodes. The nodes that are returned are those nodes that are present at any level in the first operand and are also top-level nodes in the second operand. Thus the FILTER operator uses the second operand as a "filter" that retains only selected nodes from the forest represented by the first operand. The filtering process is based on node identity—that is, it requires both operands to contain the same node, not just two nodes with the same value. Obviously, if the two operands do not have a common root, the result of the FILTER expression is empty.

The action of a FILTER expression is illustrated by Figures 3a and 3b. Figure 3a shows an ordered forest that might result from evaluating the path expression /C. Each tree is rooted in a node of type C. Figure 3b shows the result when this ordered forest is filtered by the path expression //A | //B. Only nodes of type A and B are retained, but where a hierarchic or sequential relationship exists among these nodes, the relationship is preserved.

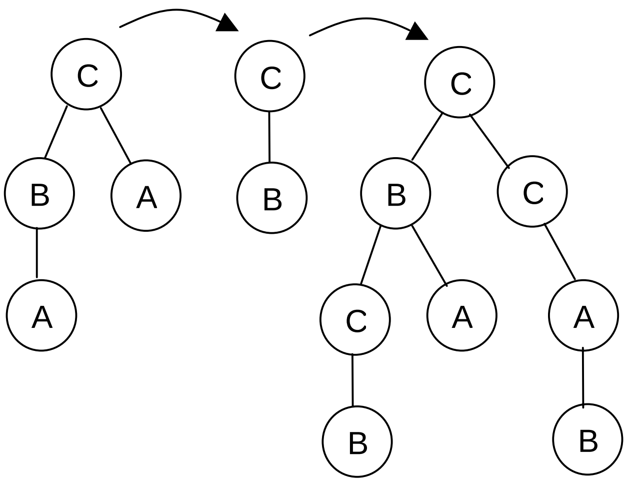

Fig. 3a. Value of `/C`

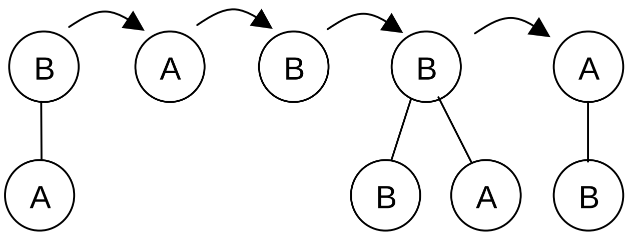

Fig. 3b. Value of `/C FILTER //A | //B`

FILTER expressions are useful in "projecting" some desired subset of a document, eliminating undesired parts while retaining the document structure. The following example illustrates this process by computing a table of contents for a document that contains many levels of nested sections. The query filters the document, retaining only section elements, title elements nested directly inside section elements, and the text of those title elements. Other elements, such as paragraphs and figure titles, are eliminated, leaving only the "skeleton" of the document.

In this example, the first operand of FILTER is the function call document( ), which returns the root of the implicit document. The second operand is a path expression that identifies the nodes to be preserved from the original document.

(Q15) Prepare a table of contents for an implicit input document, containing nested sections and their titles.

```
<toc>
   document( ) FILTER
   //section | //section/title | //section/title/text()
</toc>
```

2.5 Conditional Expressions

Conditional expressions are useful when the structure of the information to be returned depends on some condition. Of course, like all Quilt expressions, conditional expressions can be nested.

As an example of a conditional expression, consider a library that has many holdings, each described by a <holding> element with a "type" attribute that identifies its type: book, journal, etc. All holdings have a title and other nested elements that depend on the type of holding.

(Q16) Make a list of holdings, ordered by title. For journals, include the editor, and for all other holdings, include the author.

```
FOR $h IN //holding
RETURN
   <holding>
      $h/title,
      IF $h/@type = "Journal"
      THEN $h/editor
      ELSE $h/author
   </holding> SORTBY (title)
```

2.6 Functions

Quilt provides a library of built-in functions for use in queries. We have already used some of the Quilt functions, such as document, which returns the root node of a named document. The Quilt function library contains all the functions of the XPath core function library, all the aggregation functions of SQL (avg, sum, count, max, and min), and a number of other useful functions. For example, the distinct function eliminates duplicates from a collection, and the empty function returns True if and only if its argument is an empty collection.

In addition to the built-in functions, Quilt allows users to define functions of their own. In general, a Quilt query consists of a set of function definitions, followed by an expression that can call the functions that are thus defined. The scope of a function definition is limited to the query in which it is defined. Each function definition must declare the types of its parameters and result.

In another paper, we expect to define an extensibility mechanism whereby function definitions with global scope, written in various programming languages, can be added to the Quilt function library.

Some functions take scalar arguments and some take collections (sets, lists, and bags) as arguments. In general, when a collection is passed to a function that expects a scalar argument, the function returns a collection in which each element is the result of applying the function to one of the elements of the original collection.

A function may be defined recursively—that is, it may be referenced in its own definition. The next query contains an example of a recursive function that computes the depth of a node hierarchy. In its definition, the user-defined function depth calls the built-in functions empty and max.

(Q17) Using a recursive function, compute the maximum depth of the document named "partlist.xml."

```
FUNCTION depth($e ELEMENT) RETURNS integer
   {
   -- An empty element has depth 1
   -- Otherwise, add 1 to max depth of children
   IF empty($e/*) THEN 1
   ELSE max(depth($e/*)) + 1
   }
depth(document("partlist.xml"))
```

To further illustrate the power of functions, we will write a function that returns all the nodes that are "connected" to a given node by child or reference connections, and a recursive function that returns all the nodes that are "reachable" from a given node by child or reference connections. The following example uses these functions to return a connected fragment of the implicit input document.

(Q18) Find all the nodes that are reachable from the employee with serial number 12345 by child or reference connections, preserving the original relationships among the resulting nodes.

```
FUNCTION connected($e ELEMENT) RETURNS SET(ELEMENT)
   { $e/* UNION $e/@*-> }
FUNCTION reachable($e ELEMENT) RETURNS SET(ELEMENT)
   { $e UNION coalesce(reachable(connected($e))) }
document( ) FILTER reachable(//emp[serial="12345"])
```

In the above example, the reachable function invokes itself with a set of elements as argument. Since reachable maps each element to a set of elements, the result of invoking reachable on a set of elements is a set of sets of elements. This intermediate result is passed to the coalesce function, which converts a set of sets to a single set that is the union of its members.

Of course, it is possible to write a recursive function that fails to terminate for some set of arguments. In fact, the reachable function in the previous example will fail to terminate if called on an element that references one of its ancestors. At present, it is the user's responsibility to avoid writing a nonterminating function call. A mechanism to help the user stay out of trouble, such as a fixpoint operator, is a subject for further research.

2.7 Quantifiers

Occasionally it is necessary to test for existence of some element that satisfies a condition, or to determine whether all elements in some category satisfy a condition. For this purpose, Quilt provides existential and universal quantifiers. The existential quantifier is illustrated in Q19, and the universal quantifier is illustrated in Q20.

(Q19) Find titles of books in which both sailing and windsurfing are mentioned in the same paragraph.

```
FOR $b IN //book
WHERE SOME $p IN $b//para SATISFIES
   contains($p, "sailing")
   AND contains($p, "windsurfing")
RETURN $b/title
```

(Q20) Find titles of books in which sailing is mentioned in every paragraph.

```
FOR $b IN //book
WHERE EVERY $p IN $b//para SATISFIES
   contains($p, "sailing")
RETURN $b/title
```

2.8 Variable Bindings

Some queries use the same expression in more than one place. In such a case, it is sometimes helpful to bind the value of the expression to a variable so that the definition of the expression does not need to be repeated. This can be accomplished by a variable binding, which looks like the LET clause of a FLWR expression. A variable binding can be used outside a FLWR expression if it is followed by the word EVAL, which suggests that, after the variable is bound, the expression that follows the binding is evaluated. In the following example, the average price of books is a common subexpression that is bound to variable $a and then used repeatedly in the body of the query.

(Q21) For each book whose price is greater than the average price, return the title of the book and the amount by which the book's price exceeds the average price.

```
LET $a := avg(//book/price)
EVAL
   <result>
      FOR $b IN /book
      WHERE $b/price > $a
      RETURN
         <expensive_book>
            $b/title ,
            <price_difference>
               $b/price - $a
            </price_difference>
         </expensive_book>
   </result>
```

A variable binding can be used in conjunction with an element constructor to replicate some parts of an existing element, as in the following example. This example uses the XPath functions name(element), which returns the tagname of an element, and number(element), which returns the content of an element expressed as a number. When an expression inside the body of an element constructor evaluates to one or more attributes, those attributes are considered to be attributes of the element that is being constructed.

(Q22) Variable $e is bound to some element with numeric content. Construct a new element having the same name and attributes as $e, and with numeric content equal to twice the content of $e.

```
LET $tagname := name($e)
EVAL
   <$tagname>
      $e/@*,   -- replicates the attributes of $e
      2 * number($e)
   </$tagname>
```

3 Querying Relational Data

Since much of the world's business data is stored in relational databases, access to relational data is a vitally important application for an XML query language. In this section, we will illustrate the use of Quilt to access relational data by a series of examples based on a schema that is often used in relational database tutorials, containing descriptions of suppliers and parts, as shown in Figure 4. In this schema, Table S contains supplier numbers and names; Table P contains part numbers and descriptions, and Table SP contains contains the relationships between suppliers and the parts they supply, including the price of each part from each supplier.

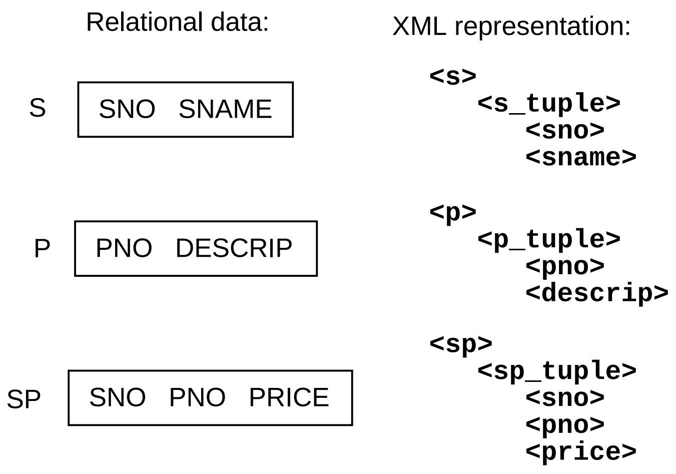

Fig. 4. One possible XML representation of relational data

Figure 4 also shows how the schema of parts and suppliers might be translated into a default XML view in which each table appears as a document, each row of a table appears as an element inside the document, and each value inside a row appears as a nested element. Other, more richly structured views can be defined on top of this default view by means of Quilt queries, as we will illustrate below.

SQL[7] is the standard relational database language. In many cases, SQL queries can be translated into Quilt queries in a straightforward way by mapping SQL query-blocks into FLWR-expressions. We illustrate this mapping by the following query:

(Q23) Find part numbers of gears, in numeric order.

SQL version:

```
SELECT pno
FROM p
WHERE descrip LIKE 'Gear'
ORDER BY pno;
```

Quilt version:

```
FOR $p IN document("p.xml")//p_tuple
WHERE contains($p/descrip, "Gear")
RETURN $p/pno SORTBY(.)
```

In Quilt, the operand of SORTBY is always interpreted within the context of the element to be sorted. Since the <pno> elements generated by Q23 have no internal structure, we use the notation "SORTBY(.)", which causes the <pno> elements to be sorted by their content.

3.1 Grouping

Many relational queries involve forming data into groups and applying some aggregation function such as count or avg to each group. In SQL, these queries are expressed using GROUP BY and HAVING clauses. The following example shows how such a query might be expressed in Quilt:

(Q24) Find the part number and average price for parts that have at least 3 suppliers.

SQL version:

```
SELECT pno, avg(price) AS avgprice
FROM sp
GROUP BY pno
HAVING count(*) >= 3
ORDER BY pno;
```

Quilt version:

```
FOR $pn IN distinct(document("sp.xml")//pno)
LET $sp := document("sp.xml")//sp_tuple[pno = $pn]
WHERE count($sp) >= 3
RETURN
   <well_supplied_item>
      $pn,
      <avgprice> avg($sp/price) </avgprice>
   </well_supplied_item> SORTBY(pno)
```

Note that $pn, bound by a FOR-clause, represents an individual part number, whereas $sp, bound by a LET-clause, represents a set of sp-tuples. The SQL HAVING clause, which applies a predicate to a set, is mapped into a Quilt WHERE-clause that operates on the set-valued variable $sp. The Quilt version of the query also uses an element constructor to enclose each part number and average price in a containing element called <well_supplied_item>.

3.2 Joins

Joins, which combine data from multiple sources into a single query result, are among the most important forms of relational queries. In this section we will illustrate how several types of joins can be expressed in Quilt.

A conventional ("inner") join returns information from two or more related tables, as illustrated by example Q25.

(Q25) Return a "flat" list of supplier names and their part descriptions, in alphabetic order.

```
FOR $sp IN document("sp.xml")//sp_tuple,
    $p IN document("p.xml")//p_tuple[pno = $sp/pno],
    $s IN document("s.xml")//s_tuple[sno = $sp/sno]
RETURN
   <sp_pair>
      $s/sname ,
      $p/descrip
   </sp_pair> SORTBY (sname, descrip)
```

Q25 returns information only about parts that have suppliers and suppliers that have parts. An "outer join" is a join that preserves information from one or more of the participating tables, including those rows that have no matching row in the joined table. For example, a "left outer join" between suppliers and parts might return information about suppliers that have no matching parts. In place of the missing parts data, relational systems usually return null values; but an XML query might represent the missing data by an empty element or the absence of an element. Q26 is an example of a Quilt query that corresponds to a left outer join.

(Q26) Return names of all the suppliers in alphabetic order, including those that supply no parts; inside each supplier element, list the descriptions of all the parts it supplies, in alphabetic order.

```
FOR $s IN document("s.xml")//s_tuple
RETURN
   <supplier>
      $s/sname,
      FOR $sp IN document("sp.xml")//sp_tuple
               [sno = $s/sno],
          $p IN document("p.xml")//p_tuple
               [pno = $sp/pno]
      RETURN $p/descrip SORTBY(.)
   </supplier> SORTBY(sname)
```

Another type of join that is sometimes used in relational systems is a "full outer join," which preserves information from both of the participating tables, including rows of each table that have no matching rows in the other table. In XML, the result of a full outer join can be structured in any of several ways. The example in Q27 uses a format of parts nested inside suppliers, followed by a list of parts that have no supplier. This might be thought of as a "supplier-centered" full outer join. A "part-centered" full outer join, on the other hand, might return a list of suppliers nested inside parts, followed by a list of suppliers that have no parts. Other forms of outer join queries are also possible.

(Q27) Return names of suppliers and descriptions and prices of their parts, including suppliers that supply no parts and parts that have no suppliers.

```
<master_list>
    (FOR $s IN document("s.xml")//s_tuple
     RETURN
        <supplier>
           $s/sname,
           FOR $sp IN document("sp.xml")//sp_tuple
                   [sno = $s/sno],
               $p IN document("p.xml")//p_tuple
                   [pno = $sp/pno]
           RETURN
              <part>
                 $p/descrip,
                 $sp/price
              </part> SORTBY (descrip)
        </supplier> SORTBY(sname)
    )
   UNION
    -- parts that have no supplier
    <orphan_parts>
       FOR $p IN document("p.xml")//p_tuple
       WHERE empty(document("sp.xml")//sp_tuple
             [pno = $p/pno] )
       RETURN $p/descrip SORTBY(.)
    </orphan_parts>
</master_list>
```

Q27 uses an element constructor to enclose its output inside a <master_list> element. The UNION operator, when used as in Q27 to combine two ordered lists, returns the first list with the second list appended at the end. The result is a <master_list> element containing an ordered list of <supplier> elements followed by an <orphan_parts> element that contains descriptions of all the parts that have no supplier.

3.3 Defining Structured Views

An application might prefer a structured XML view of the database of parts and suppliers, such as the "master_list" generated by Q27, rather than a simpler view in which each table appears as a separate document. If a relational database system can present simple default XML views of its tables, the job of constructing more structured views can be left to Quilt. Just as in SQL, a Quilt query can serve as the definition of a persistent view of underlying data. For example, by means of suitable data definition statements, Q27 could be entered into the system catalogs as the definition of a persistent XML view called "master_list". Quilt queries against the master_list could then be automatically merged with the view-definition to form queries against the underlying tables.

4 Conclusion

With the emergence of XML, the distinctions among various forms of information, such as documents and databases, are quickly disappearing. Quilt is designed to support queries against a broad spectrum of information sources by incorporating features from several languages that were originally designed for diverse purposes. This paper has illustrated the versatility of Quilt by using it to express queries against both semi-structured documents and relational databases. We believe that Quilt represents a promising approach to a query language that can help to realize the potential of XML as a universal medium for data interchange.

5 Acknowledgements

The authors thank Mary Fernandez, Phil Wadler, Jerome Simeon, and Zack Ives for many helpful discussions during the definition of the Quilt language.

Appendix: Quilt Grammar

The following rather permissive grammar will be augmented by a set of typing rules (for example, an expression used in a predicate must return a Boolean value).

```
QuiltQuery ::= NamespaceDeclList? FunctionDefnList? Expr
NamespaceDeclList ::= NamespaceDecl*
NamespaceDecl ::= 'NAMESPACE' Identifier '=' StringLiteral
FunctionDefnList ::= FunctionDefn*
FunctionDefn ::=
      'FUNCTION' FunctionName '(' ParamList? ')' 'RETURNS' Datatype '{' Expr '}'
FunctionName ::= QName
ParamList ::= Parameter ( ',' Parameter )*

/* The name and type of each parameter are declared */
Parameter ::= Variable Datatype
Datatype ::= PrimitiveDatatype
   | Collection '(' Datatype ')'

/* Primitive datatypes will be expanded to all Schema primitive types,
   plus ELEMENT to denote a generic element. We avoid the keyword STRING
   because it would pre-empt the name of a function used in XPath. */
PrimitiveDatatype ::= 'CHARSTRING' | 'INTEGER' | 'ELEMENT'
Collection ::= 'SET' | 'LIST' | 'BAG'
Expr ::= LogicalExpr
   | LogicalExpr 'SORTBY' '(' SortSpecList ')'
```

```
SortSpecList ::= SortSpec
    | SortSpecList ',' SortSpec
SortSpec ::= Expr
    | Expr 'ASCENDING'
    | Expr 'DESCENDING'
LogicalExpr ::= LogicalTerm
    | LogicalExpr 'OR' LogicalTerm
LogicalTerm ::= LogicalFactor
    | LogicalTerm 'AND' LogicalFactor
LogicalFactor ::= FilteredExpr
    | 'NOT' FilteredExpr
FilteredExpr ::= SetExpr
    | FilteredExpr 'FILTER' SetExpr
SetExpr ::= SetTerm
    | SetExpr 'UNION' SetTerm
SetTerm ::= SequencedValue
    | SetTerm 'INTERSECT' SequencedValue
    | SetTerm 'EXCEPT' SequencedValue
SequencedValue ::= QuiltValue
    | SequencedValue 'BEFORE' QuiltValue
    | SequencedValue 'AFTER' QuiltValue
QuiltValue ::= Comparison
    | SpecialExpr
Comparison ::= ArithExpr
    | Comparison CompareOp ArithExpr
ArithExpr ::= ArithTerm
    | ArithExpr '+' ArithTerm
    | ArithExpr '-' ArithTerm
ArithTerm ::= ArithFactor
    | ArithTerm '*' ArithFactor
    | ArithTerm 'DIV' ArithFactor
    | ArithTerm 'MOD' ArithFactor
ArithFactor ::= ArithPrimitive
    | '+' ArithPrimitive
    | '-' ArithPrimitive
ArithPrimitive ::= BasicExpr Predicate*
    | DisjointPathExpr
DisjointPathExpr ::= PathExpr
    | DisjointPathExpr '|' PathExpr
PathExpr ::= RegularExpr
    | '/' RegularExpr
    | '//' RegularExpr
    | BasicExpr Predicate* '/' RegularExpr
    | BasicExpr Predicate* '//' RegularExpr
```

```
/* For now, we support only the '/' and '//' operators of XPath.
   In the future, we may support other forms of regular expressions. */
RegularExpr ::= Step Predicate*
   | RegularExpr '/' Step Predicate*
   | RegularExpr '//' Step Predicate*

/* As in XPath, a step represents movement in an XML document along the
   child, parent, or attribute axis.  If followed by '->', the step
   dereferences a key or IDREF attribute and returns the target element
   (this requires information from a schema or Document Type Definition.) */
Step ::= NameTest
   | NodeType '(' ')'
   | '@' NameTest
   | '..'
   | Step '->'

/* An expression in a predicate must evaluate to a Boolean or an ordinal number */
Predicate ::= '[' Expr ']'
   | '[' 'RANGE' Expr 'TO' Expr ']'
BasicExpr ::= Variable
   | Literal
   | FunctionName '(' ExprList? ')'
   | '(' Expr ')'
   | NodeConstructor
   | '.'
Literal ::= StringLiteral
   | IntegerLiteral
   | FloatLiteral
ExprList ::= Expr
   | ExprList ',' Expr
SpecialExpr ::= LetClause 'EVAL' QuiltValue
   | FlwrExpr
   | 'IF' Expr 'THEN' QuiltValue 'ELSE' QuiltValue
   | Quantifier Variable 'IN' Expr 'SATISFIES' QuiltValue
Quantifier ::= 'SOME'
   | 'EVERY'
FlwrExpr ::= ForLetClause WhereReturnClause

/* A ForLetClause has at least one ForClause and any number
   of additional ForClauses and LetClauses */
ForLetClause ::= ForClause
   | ForLetClause ForClause
   | ForLetClause LetClause
ForClause ::= 'FOR' Variable 'IN' Expr
   | ForClause ',' Variable 'IN' Expr
```

```
LetClause ::= 'LET' Variable ':=' Expr
    | LetClause ',' Variable ':=' Expr
WhereReturnClause ::= WhereClause? ReturnClause
WhereClause ::= 'WHERE' Expr
ReturnClause ::= 'RETURN' QuiltValue

/* For now, a node constructor is an element constructor.
    In the future we will add processing instructions and comments. */
NodeConstructor ::= ElementConstructor
ElementConstructor ::= StartTag ExprList? EndTag
    | EmptyElementConstructor
StartTag ::= '<' TagName AttributeList? '>'
TagName ::= QName
    | Variable
AttributeList ::= (AttributeName '=' ArithExpr)*
AttributeName ::= QName
    | Variable
EndTag ::= '</' TagName '>'
EmptyElementConstructor ::= '<' TagName AttributeList? '/>'

/* A name test is a Qname where "*" serves as a wild card. */
NameTest ::= QName
    | NamePrefix ':' '*'
    | '*' ':' LocalPart
    | '*'
NodeType ::= 'NODE'
    | 'TEXT'
    | 'COMMENT'
    | 'PROCESSING_INSTRUCTION'
QName ::= LocalPart
    | NamePrefix ':' LocalPart
NamePrefix ::= Identifier
LocalPart ::= Identifier
CompareOp ::=  '=' | '<' | '<=' | '>' | '>=' | '!='
```

The terminal symbols of this grammar (other than keywords and special symbols) are:

Variable	(example: $x)
Identifier	(example: x)
StringLiteral	(example: "x")
IntegerLiteral	(example: 5)
FloatLiteral	(example: 5.7)

As in XPath, many Quilt operators are overloaded and can be applied to values of various types. For example, A = B where A and B are sets is true if and only if there exists an element a in A and an element b in B such that a = b. The detailed semantics of these operators will be provided as part of a more complete language specification.

The Quilt core function library includes the following:

1. The functions of the XPath core function library[4]
2. The "aggregation functions" of SQL that operate on a collection and return a scalar result: sum, count, avg, max, and min.
3. Additional functions such as the following (partial list):
   document(string) returns the root node of a named document
   empty(collection) returns True if its argument is empty
   distinct(collection) removes duplicates from its argument
   name(element) returns the name (generic identifier) of an element
   shallow(element) strips an element of its subelements
   coalesce(set(set(element))) converts a set of sets into a single set that is the union of its members

References

1. World Wide Web Consortium. *Extensible Markup Language (XML) 1.0.* W3C Recommendation, Feb. 10, 1998. See http://www.w3.org/TR/1998/REC-xml-19980210
2. World Wide Web Consortium. *XML Schema, Parts 0, 1, and 2.* W3C Working Draft, April 7, 2000. See http://www.w3.org/TR/xmlschema-0, -1, and -2.
3. World Wide Web Consortium. *XML Query Requirements*. W3C Working Draft, Jan. 31, 2000. See http://www.w3.org/TR/xmlquery-req
4. World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, Nov. 16, 1999. See http://www.w3.org/TR/xpath.html
5. J. Robie, J. Lapp, D. Schach. *XML Query Language (XQL).* See http://www.w3.org/TandS/QL/QL98/pp/xql.html.
6. Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *A Query Language for XML.* See http://www.research.att.com/~mff/files/final.html
7. International Organization for Standardization (ISO). *Information Technology—Database Language SQL*. Standard No. ISO/IEC 9075:1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.)
8. Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann Publishers, San Francisco, 1996.
9. Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries,* 1(1):68-88, April 1997. See http://www-db.stanford.edu/~widom/pubs.html
10. S. Cluet, S. Jacqmin, and J. Simeon. *The New YATL: Design and Specifications.* Technical Report, INRIA, 1999.

11. Jonathan Robie, Don Chamberlin, and Daniela Florescu. Quilt: an XML Query Language. Graphic Communications Association, *Proceedings of XML Europe*, June 2000.

12. World Wide Web Consortium. *XML Query Data Model*. W3C Working Draft, May 11, 2000. See http://www.w3.org/TR/query-datamodel.

13. World Wide Web Consortium. *XSL Transformations (XSLT)*. W3C Recommendation, Nov. 16, 1999. See http://www.w3.org/TR/xslt.

14. World Wide Web Consortium. *XML Pointer Language (XPointer)*. W3C Working Draft, Dec. 6, 1999. See http://www.w3.org/TR/WD-xptr.

15. World Wide Web Consortium. *Namespaces in XML*. W3C Recommendation, Jan. 14, 1999. See http://www.w3.org/TR/REC-xml-names.