Reihe Informatik
16 / 2000

YAXQL:
A powerful and web-aware query language supporting query reuse and data integration

Guido Moerkotte Thorsten Fiebig

YAXQL:

A powerful and web-aware query language supporting query reuse and data integration

Guido Moerkotte and Thorsten Fiebig

Fakultät für Mathematik und Informatik

University of Mannheim

68131 Mannheim

Germany

(moerkotte | fiebig)@informatik.uni-mannheim.de

August 7, 2000

Abstract

Since XML seems to be the next great wave on the web, several query languages for XML have been proposed. Unfortunately, none of these proposals comes even close to meet the requirements for such a query language. We review the requirements for a query language for XML and propose a new query language, YAXQL, which meet them.

1 Introduction

XML [6] is the first semistructured data model of practical relevance and soon there will be a need for a query language for querying XML documents. Research offers query languages from two areas: query languages for the web (like W3QL [22], WebSQL [27, 28] and WebOQL [4] that build on SQL and OQL or logic-based languages like WebLog [23] and FLORID [20]), and query languages for semistructured data (like Lorel [3][1], OQL-doc [2], StruQL [17], UnQL [7] and YATL [10, 11]). Query languages for the web concentrate on extracting information from web pages, typically written in HTML. Since HTML tags do not carry document or application specific semantics but specify some document structure and mostly layout instead, these query languages concentrate on extracting data from documents by pattern and structure matching. While these capabilities remain useful for XML, these query languages cannot deal with the specific features of XML documents. Query languages for semistructured data build

[1] which was later adapted to XML[19]

on their own data model. This data model often does not incorporate order—an essential feature of XML documents. Further, they neglect the details of XML and therefore do not support its essential features. However, some of these query languages have been extended to deal better with XML documents [19]. The query languages proposed so far have not meet the requirements stated in the next section yet.

Lately, several proposals for query languages for XML were presented at a W3C workshop [26] on query languages for XML. Even the most promising proposals (like XQL [30], XQuery [15, 14], <unnamed> [5]) do not extend XPath [9]—a language recommended by W3C for addressing parts in XML documents—to a full fledged pattern matching language. These extensions also fail to meet our requirements. Whereas the above mentioned query languages have their roots in document processing, the database community so far proposed three query languages to deal with XML documents. XML-QL [16] derived from StruQL [17]—together with Lorel [3]—is one of the first query languages proposed. Both lack essential XML support like dealing with IDREFS lists. Further, they do not meet the requirements discussed in the next section. A newer proposal is YATL [10, 11]. It has the same deficiencies as the other two languages. Furthermore, as shown in [18], these query languages are mainly syntactic variations of each other. Whereas these query languages follow the traditional select-from-where paradigm—although with syntactic variations—UnQL [7] goes its own way by applying structural recursion to query documents. UnQL does not have special support for XML and its underlying data model, an edge labeled graph, is different from the underlying data model for XML (see the XPath documentation which discusses XML's data model [9]).

One of the most important outcomes of the W3C query language workshop [26] is a list of requirements [12]. Unfortunately, this list contains requirements that are not central and some important requirements are missing. Hence, we compiled a list of requirements that are in our opinion core requirements for a query language for XML. This list is discussed in the next section. Section 3 gives an informal introdcution to YAXQL by describing its basic features. Section 4 demonstrates the query reuse and data integration capabilities of YAXQL by means of example queries. Since—due to space limitations—not all features of YAXQL can be presented by examples, section 6 briefly discusses some more features. One of the unique features of YAXQL is its possibility to have parameters that may be unbound at query evaluation time. To support this feature, a 4-valued logic is sketched in section 5. Section 7 concludes the paper. Appendix A contains sample DTDs and documents underlying the example queries. Appendix B contains most of YAXQL's DTD. **The latter appendix is included for reviewing purposes only.**

2 Hypothesis and Requirements

We start by giving four hypotheses that drove the development of the requirements and YAXQL. The first hypothesis is that (H 1) query languages are good in processing variable bindings. This is in our opinion the most prominent feature of existing query languages. A feature we want to keep. The second hypothesis (H 2) states that a query should consist of four parts. The responsibilities of these parts are

1. to produce variable bindings,

The query language must ...

1. ... be web-aware;
2. ... provide for maximal reuse of queries;
3. ... allow explicit type conversions;
4. ... be sufficiently expressive;
5. ... be able to construct XML documents;
6. ... allow queries to return not only values but also references (e.g. XPointers);
7. ... build upon other standards: XPath [9], XPointer [25], and XSLT [8].

Figure 1: Classes of Core Requirements

2. to restrict the collection of produced variable bindings,
3. to project the relevant variable bindings, and
4. to construct the result.

Note that the third and fourth part coincide in traditional query languages like SQL and OQL. However, we want to keep these parts separate (by providing different syntactic constructs) in order to be able to reuse parts of a query. More specifically, we want to reuse the part of the query that produces variable bindings and separate it from the result construction. This ability will be one of our core requirements. Reuse of queries implies that queries are identifiable entities. All query languages mentioned in the introduction fail to support query identification and query reuse. To compensate this deficiency, some proposals exist for view mechanisms [1].

Hypothesis 3 (H 3) concerns the first part of a query. It states that tree matching is the preferred way to produce variable bindings from a (set of) XML document(s). The requirement directly following from this hypothesis is that any query language for XML should provide a full-fledged support for tree pattern matching. All query languages proposed so far fail to do this. (For a good overview of different tree-matching classes and their complexity see [21]).

Hypothesis 4 (H 4) states that a query language for XML must be web-aware. This has several implications. First, it must allow to query documents that are not only contained in a given repository for semistructured data, but also any document contained anywhere in the Web. Further, if data services are provided by a forms-based interface, the query language must support querying these interfaces. The query languages for semistructured data proposed so far fail to meet these requirements.

Fig.1 contains some classes of core requirements for any query language for XML. Every class contains a whole bunch of requirements. The most important ones will be discussed below. Some of the requirements are contained in [12, 29, 24]. The first requirement directly turns our fourth hypothesis into a requirement. We also touched the second requirement. It is partially a consequence of web-awareness but has several implications. Identification and referenciability of queries to maximize possible reuse of queries are two core implications. In order to find interesting queries which can be reused, the syntax of the query language should be XML in order to allow for querying queries

homogeneously. This requirement can be motivated as follows. It is very likely that data providers do not provide unlimited access to static data collections but instead provide access to the data collections via services. These services can easily be specified by a query language. A value-added service could combine several of these sources by joining data from different sources. (This feature will be highlighted by some of our example queries.) In order to do so, queries must be identifiable, reusable units that can be referenced within other queries. Note that view mechanisms fail at this point. If only views and not queries can be used within a query definition, the service providers are forced to specify any service twice: once as a query and once as a reusable view.

There are more implications to be drawn from the requirement to maximize reuse. Consider a service that is provided by a forms-based interface. Not necessarily all the entries in the forms have to be filled in by the user. Nevertheless, we want the service provider to be able to specify only a single query instead of one query for every possible combination of filled entries[2]. This kind of reuse requires the query to contain several free variables (parameters/arguments) and only some of them are bound prior to query evaluation. This implies to modify the underlying logic of query languages seen so far. We tackle this problem in section 5 where we introduce a four-valued logic specifically tailored to deal with the semantics of queries containing free variables. None of the query languages proposed so far considers this point.

Requirement 3 is motivated easily by the observation that both "10" < "3" and 3 < 10 are valid. Requirement 4 is partially obvious and will partially be motivated by our example application in the next section. It implies support of joins, nested queries, recursive queries, full-text search, and the like. It also includes query support for all XML features. Consider a simple example of the latter. XML provides an attribute type `IDREFS`. Such an attribute contains a blank separated list of element identifiers. None of the query languages proposed so far (not even XQL [30], XQuery [15, 14], <unnamed> [5], XML-Query [16]) provides a mechanism to access these identifiers separately, e.g. by iterating through them.

A closure requirement is not necessarily a feature of a query language for XML (see also the discussion in [24]). Instead, the query language can be restricted to produce variable bindings. The part then converting a query result into a representation (e.g. an XML document (see Requirement 5)) can also be implemented by using a style sheet language. This is essentially the approach taken by YAXQL. However, we need to provide some glue mechanisms to group variable bindings[3] and to iterate through them. In all current query languages, variables are bound to (copied) XML-fragments (including flat text), which makes it impossible to return a list of X-Pointers to XML-fragments. We think that it is absolutely necessary for a query language to allow variables to be bound to X-Pointers to provide a reference mechanism and avoid copy semantics in cases where this is not desired. Requirement 6 suggests to use the X-Pointer [25] mechanism.

Obviously, other work that is related to a query language for XML should be incorporated. This point is stressed by requirement 7.

[2]For n parameters there are 2^n possible bound/unbound combinations. Stating all these explicitly is rather tedious

[3]Some query languages express grouping with Skolem functions.

Queries on the web will be used by millions of users but these users will comparatively seldom pose ad-hoc queries[4]. Hence, terseness is no requirement.

2.1 Scenario

A variety of databases for biological applications exists on the Web. The servers that we are interested in are the Expasy server (http://www.expasy.ch), the EBI server (http://srs.ebi.ac.uk), and the Genome server (http://www.genome.ad.jp). The Expasy server contains the standardized enzyme classification. The EBI server contains detailed enzyme information and the genome server contains detailed information about reactions catalyzed by enzymes. None of these servers provide their data in XML. To illustrate our example queries, we extracted the essence of these servers and designed our own DTDs. Together with sample documents, these DTDs are listed in appendix A. We assume that all these servers provide some services. Currently, these servers provide these services as a form-based query interface. We will mimic these services by casting the underlying queries into YAXQL queries. For the service provider, using a declarative language to specify the services has the usual advantages.

Since any of these servers provides only a limited view of the data around enzymes, we will further open a new server that provides value-added services by integrating the data. We will call this server NEW. For the other servers we use the following names:

EC hierarchical enzyme classification

EDB detailed enzyme information data base

RDB reaction database

For every server we assume that all local queries are contained in a file queries.xml managed at the server site.

3 YAXQL Basics

This section gives an informal introduction to YAXQL. As a YAXQL query is a combination of XML elements it is valid XML data. Hence, we do not use the usual terms like clause or operator for the introduction of a query language, but the term element.

For the structuring of queries YAXQL comes along with several top-level elements. Members of the top level elements are `xql:query`, `xql:construct` and `xql:query-construct`. The `xql:query` element includes the first three parts of H 2. The `xql:construct` element describes the result construction. To combine the two elements YAXQL provides the `xql:query-construct` element.

The remainder of the section is structured as follows. First, the generation of variable binding is described. Therefore, we introduce several variable binding elements. Next, we introduce the processing of variable bindings in YAXQL. Finally we sketch the result construction in YAXQL. The query reuse capabilities of YAXQL are described in the following section.

[4]This situation is similar to the situation in the relational context where end-users rarely pose SQL queries.

3.1 Basic Variable Binding Elements

In YAXQL a binding of a variable is a mapping of a variable name to its associated value. So, it is represented as a pair containing the variable name and the bounded value. Bindings of different variables are combined to a variable binding set. There are three different value types: nodes, strings and X-Pointers. Thus, we define the three different binding types `source`, `string` and `X-Pointer`. In the first case, the XML fragment is copied and assigned to the variable. In the second case, the XML fragment is extracted and all the markup is eliminated according to the rules specified for the `string()` function in XPath. In the third case, the variable is bound to an X-Pointer [25], that points to the specified data.

YAXQL features the basic variable binding elements `xql:match` and `xql:bind`. A variable binding element declares a single variable and creates bindings for the variable by matching an extended X-Path expressions. As these expressions are already described in [9] we do not give a detailed description here. The result of the binding process described by the elements is a collection of variable binding sets.

The variable binding elements provide the attributes `var`, `data-type`, `select`, `href` and `bind-to`. The `var` attribute specifies the name of the variable. The optional attribute `data-type` defines a default data type of the variable binding. This defines a default data conversion that is applied each time the value of a binding is accessed. The content of the `select` attribute is the extended X-Path expression that references the relevant nodes in the XML document. The X-Path expression references nodes relative to a given start node. In a variable binding element there are two possibilities for the specification of a start node. First, a URI of an XML document can be specified in the `href` attribute. Then the root node of this document becomes the start node of the X-Path expression. Second, a value of a given variable binding can be used. But only the value of variable bindings with the binding type `source` or `X-Pointer` can be used as a start node. In order to use the value of a `source` binding the X-Path expression has to be preceded by the variable name. The X-Path concatenation `"/"` glues the variable name and the X-Path expression. If the binding has the type `X-Pointer` the operator `"/"` concatenates the variable name and the X-Path expression. The `bind-to` attribute determines the binding type discussed above.

The following query gives an example for the application of the variable binding elements. It queries the EDB server mentioned in our scenario. It produces a collection of variable binding sets, by application of a combination of a `xql:match` and a `xql:bind` element. Each set contains the bindings for the variables `enzyme` and `ph-opt`. Thus, the collection lists enzymes together with their optimal ph value:

```
<xql:query name="example-query">
     <xql:match var="enzyme"
                href="edb.xml"
                select="/edb:enzymes/edb:enzyme"
                bind-to="source"/>
     <xql:bind var="ph-opt"
               data-type="number"
```

```
            select="$enzyme/edb:ph-opt"
            bind-to="string"/>
</xql:query>
```

The `xql:match` element binds the variable `enzyme` to the enzyme nodes found in the document `edb.xml` by the X-Path expression `/edb:enzymes/edb:enzyme`. The result is a bag of sets whose elements represent the bindings of the `enzyme` variable.

In contrast to `xql:match`- the `xql:bind` element only creates a single binding for its variable. Therefore it takes the first according to the document order of the node set described by the X-Path expression. Thus, the result is a set that contains a single variable binding set. In our example query the `xql:bind` creates for each binding of the `enzyme` variable a string binding for the `ph-opt` variable. Hence, the `xql:bind` contains a simple X-Path expression that is preceded by the variable `enzyme`.

The result of the combined variable binding element is constructed as follows. For each variable binding set of the `xql:match` element we create a set that has the variable binding set as its only element. Next, we build the union of this set and the variable binding set that is returned by the `xql:bind` element. All union sets together form our result.

In order to retreive the value of variable bindings YAXQL provides the elements `xql:value` and `xql:deref`. The `xql:value` returns the value of a binding. If the binding type is `source` or `string` and a default data type is given, `xql:value` converts the value accordingly. In contrast to `xql:value` the `xql:deref` element can only be applied to `X-Pointer` bindings. The element dereferences the given X-Pointer and returns the result. If a default data type is given, the result is converted accordingly. Both elements have the optional attribute `data-type`. if there is no given default data type, it can be used for the explicit specification of a data type.

Moreover, the `xql:source` can be used to access the value of a variable binding. Like `xql:bind` it selects a single node by applying an extended X-Path expression, but without creating variable bindings.

For the expression of tree patterns we use nested `xql:match` elements. A nested `xql:match` element has a parent element. Its content consists of several child elements. The semantic of the nesting is that the child nodes of the variable binding of the parent element are the start nodes of the X-Path expressions of the child elements. To achieve a greater flexibility for the expression of tree patterns the `xql:match` provides the attributes `ordered` and `max-gap`. If the order of the child elements should coincide with the order of their matching in the document, the `order` attribute of the parent element is set to `YES`. The `max-gap` attribute defines the maximal distance between the start nodes of two neighbor child elements.

3.2 Variable Binding Processing

By the Processing of variable bindings we mean filtering and projection. Consider the following extension of our example query:

```
<xql:query name="example-query">
```

```
<xql:match var="enzyme"
           href="edb.xml"
           select="/edb:enzymes/edb:enzyme"
           bind-to="source"/>
<xql:bind var="ph-opt"
          data-type="number"
          select="$enzyme/edb:ph-opt"
          bind-to="string"/>
<xql:predicate>
     <xql:lt>
          <xql:value var="ph-opt">
          <xql:constant data-type="number"> 7 </xql:constant>
     </xql:lt>
</xql:predicate>
<xql:project>
     <xql:bind var="enzyme-class"
               select="$enzyme/edb:ec-class"
               bind-to="string"/>
</xql:project>
</xql:query>
```

The query generates a bag of variable binding sets that consist of a single element representing the binding of the `enzyme-class` variable. The value of the binding is a string containing the class name of an enzyme that has an optimal ph-value that is less then seven.

In YAXQL the filtering of variable binding is performed by the application of boolean expressions. The expressions are embedded in the content of a `xql:predicate` element. The elements `xql:and`, `xql:or` and `xql:not` represent the boolean operators *and*, *or* and *not*. The content of the `xql:and` and the `xql:or` element is a list of boolean expressions, which are based on the usual boolean functions. As in Lisp the YAXQL comparison operators can be applied on a list of operands. Moreover, there are elements to express all and exist quantifiers. The evaluation of a boolean expression is based on four-valued logic, which is needed for the evaluation of expressions with unbound free variables. Further details are discussed in section 5.

In our example query we access the binding of the `ph-opt` variable by the application of a `xql:value` element. The filtering predicate is a less-then comparison between the value of the `ph-opt` binding and the constant expression 7.

For the utilization of constant expressions in YAXQL we introduce the `xql:constant` element. In order to reuse a constant expression the element has the `const` attribute. Its value contains a unique identifier for the expressions. Additionally the element has a `data-type` attribute, which specifies a data type of the constant expression.

The final part of the variable processing is the projection which determines the structure of the query result. For the projection we introduce an `xql:project` element that contains several `xql:bind` elements. These elements are applied to every filtered variable binding set. The union of the resulting variable binding sets forms an element of the query result.

In our example query the filtered variable binding sets are projected to the bindings of the variable `enzyme-class`. which is bound to the class name of the enzymes referenced by the variable `enzyme`.

3.3 Result Construction

Result construction transforms a given collection of sets of variable bindings into some textual form. This will mostly be an XML document but can also be something else like an HTML document. The `construct` element specifies a result. If the output of the `construct` element is known to conform to some DTD, this DTD can be specified. Any style sheet language like XSL will be very helpful to construct XML documents. Given such a style sheet language, we only need some glue mechanisms to transform our variable bindings into text.

Consider the following example of a result construction. It is based on the variable bindings of our example query. The result consists of a single `enzymes` element which contains an `enzyme` element for each enzyme class. The content of the `enzyme` element is the string representation of the name of the class the enzyme is belonging to.

```
<xql:construct name="example-construct">
  <enzymes>
    <xql:for-each vars="enzyme-class">
      <enzyme> <xql:value var="enzyme-class"/></enzyme>
    </xql:for-each>
  </enzymes>
</xql:construct>
```

For the result construction all non-YAXQL elements are placed in the result. In order to include elements based on the input variable bindings we introduce the `xql:for-each` element. The element describes a grouping of the input variable bindings according to the variables referenced by the `vars` attribute. For each group the element content is embedded into the result. For the embedding of variable bindings into the query result we use the elements `xql:value`, `xql:deref` and `xql:source`.

4 Query Reuse and Data Integration in YAXQL

In this section we demonstrate the query reuse and data integration capabilities of YAXQL. These capabilities are based on the modularization of YAXQL. To achieve this we separate the generation and the processing of variable bindings from the result construction. The prerequisite for the reuse are unique identifiers for the different query parts. Therefore the YAXQL top elements are featured with a name attribute that contains a unique identifier.

Another important point is the possibility to define free variables in an `xql:query` or in an `xql:construct` element. A free variable is declared by the `xql:declare` element. The element attributes `var` and `data-type` determine the name and the data-type. The `maybe-unbound` attribute determines whether the query part can be evaluated without binding the free variable.

A `xql:query` element is referenced by an `xql:query-reference` element. The referenced element is specified by the value of the `href` attribute. The content consists of several `xql:bind` elements that bind the free variables of the referenced query. Moreover, the content may contain an `xql:predicate` element to filter the variable bindings of the referenced query and an `xql:project` to customize their structure.

For the referencing of an `xql:construct` element a `xql:construct-reference` element is used. Like `xql:query-refe` the referenced element is specified by the value of the `href` attribute. Not only `xql:query` and `xql:construct` elements can be referenced, but also a combination can be referenced by a `xql:query-construct` element.

In the remainder of this section we give some examples for the reuse of YAXQL queries. First, we show how free variables are declared in `xql:query`-elements. Second, we describe the referencing of `xql:query` element and show how arguments are passed and how even queries can be used as arguments. Finally, we give an example query that demonstrates the data integration capabilities of YAXQL.

4.1 Declaration of Free Variables in YAXQL

The EC server provides a service to look up an enzyme class given the classification number. This service can be implemented by a query with one free variable (parameter/argument) that will be bound to the classification number at query evaluation time. The query is

```
<xql:query name="ec:q1">
    <xql:declare var="ec-class" maybe-unbound="NO"/>
    <xql:project>
        <xql:bind var="result"
                  href="ec.xml"
                  select="/ec:classes/*[@name=$ec-class]"
                  bind-to="X-Pointer"/>
    </xql:project>
</xql:query>
```

Another example query with free variables queries the EDB server that contains detailed information about enzymes. Among the data recorded for every enzyme is its systematic name, its recommended name and alternative names. A specific service allows to look up enzymes by any of these names. The following query uses disjunction ("|"[5]) in the selector predicate to search for any name given as a parameter to the query.

```
<xql:query name="edb:q2">
```

[5] see XPath [9]

```
    <xql:declare var="ec-name" maybe-unbound="NO"/>
    <xql:project>
        <xql:bind var="result"
                  href="edb.xml"
                  select="/edb:enzymes/edb:enzyme/[@sysname=$ec-name |
                                                  @recname=$ec-name |
                                                  @name=$ec-name]"
                  bind-to="X-Pointer"/>
     </xql:project>
</xql:query>
```

4.2 Reuse of YAXQL Queries

After the introduction of queries with free variables we are ready to give an example for query reuse. In order to uniquely access a variable v bound by a query q, we use the notation $\$(q)v$. Hence, `$(ec:q1)result` references the *result* variable of query $q1$. The following query retrieves all enzymes belonging to a given class. Since the enzyme classification is hierarchical and no level is specified, one must descend arbitrarily deep. We use the Kleene operator "**"[6] in conjunction with the dereference operator to follow an arbitrary path of `ec:links`. To determine the starting point, we reuse query `ec:q1` which retrieves the ec-class we are interested in.

```
<xql:query name="ec:q3">
    <xql:declare var="ec-class-name" maybe-unbound="NO"/>
    <xql:query-reference href="ec:q1">
        <xql:bind var="ec-class" select="$ec-class-name"/>
    </xql:query-reference>
    <xql:match var="enzyme"
               select="$(ec:q1)result->/(ec:link/@href->)**/ec:enzyme"
               bind-to="X-Pointer"/>
    <xql:project>
        <xql:bind var="result" select="$enzyme"/>
    </xql:project>
</xql:query>
```

The following query against the RDB server gives a more sophisticated example of query reuse. Given a set of enzymes, specified by a *query* and its result variable name, the enzymes and their ph-values are retrieved and ordered by the ph-value. The parameter query must be evaluable without specifying any argument.

```
<xql:query name="rdb:q2">
```

[6] "*" denotes a wild card in XSL-T. Hence, we use "**" to denote the Kleene operator.

```
  <xql:declare var="query"/>
  <xql:declare var="result-variable"/>
  <xql:query-reference href="$query"/>
  <xql:match var="ph" select="$($attr)/edb:ph-opt/" bind-to="string"/>
  <xql:project order-by="ph-value ASC">
      <xql:bind var="enzyme" select="$($query).($result-variable)"/>
      <xql:bind var="ph-value" select="$ph"/>
  </xql:project>
</xql:query>
```

4.3 Data Integration

For the demonstration of the data integration capabilities of YAXQL we introduce a example query of our value adding NEW server. The query retrieves all enzyme data sheets of a class of enzymes contained in EDB by first accessing all enzyme names of the class via query `ec:q3` and then evaluating `edb:q2` for every retrieved enzyme name.

```
<xql:query name="new:q1">
    <xql:declare var="ec-class-name" maybe-unbound="NO"/>
    <xql:query-reference href="http://www.ec.bio/queries.xml/id(ec:q3)">
        <xql:bind var="ec-class" select="$ec-class-name"/>
    </xql:query-reference>
    <xql:query-reference href="http://www.edb.bio/queries.xml/id(edb:q2)">
        <xql:bind var="ec-name" select="$(ec:q3)result->/ec:name/text()"/>
    </xql:query-reference>
    <xql:project>
        <xql:bind var="result" select="$(edb:q2)result"/>
    </xql:project>
</xql:query>
```

5 4-Valued Logic for unbound variables

A YAXQL query can have several free variables. Prior to query evaluation, some of them can be bound, others may be left unbound, if the query allows so. The semantics of a query with unbound variables is given by the truth tables for a 4-valued logic. The values are T for true, F for false, U for unknown , I for ignore. Any boolean term with an unbound variable will evaluate to I. Boolean terms having variables bound to NULL will evaluate to U. Variables may be NULL due to optional matches in case there is no match. This case must clearly be distinguished from a variable being unbound. The truth tables are:

$\wedge$	T	F	U	I
T	T	F	U	T
F	F	F	F	F
U	U	F	U	U
I	T	F	U	I

$\vee$	T	F	U	I
T	T	T	T	T
F	T	F	U	F
U	T	U	U	U
I	T	F	U	I

$\neg$	T	F	U	I
	F	T	U	I

For a predicate in a query: a set of variable bindings qualifies whether the predicate evaluates to either I or T. Note that the boolean connectors are commutative and associative. Further, DeMorgan's Law holds. Distributivity does not hold, if one of the inner arguments is I. For example, $x \wedge (y \vee z) \equiv (x \vee y) \wedge (x \vee z)$ does not hold if y or z is I and x is not I.

6 Advanced concepts in YAXQL

After we have seen several examples of queries in YAXQL, we now describe some more concepts of YAXQL not illustrated by examples. Among these concepts are more possibilities to produce variable bindings (among them one for accessing legacy relational database systems), accessing web-sites exhibiting a form-based interface, defining user defined functions and predicates, conditional expressions, glue for defining documents with form-based interfaces with a YAXQL query as the evaluation mechanism, a grouping-clause and a having-clause, existential and universal quantifiers, and several set and join operations including outer-joins.

YAXQL provides several variable binding mechanisms. The most important one is `xql:match` which can be nested to build arbitrary tree patterns. Attributes allow to specify the relevance of order among child matches and can limit gaps between matching children.

Another useful binding mechanism is provided by the `xql:iterate` element that iterates through lists of strings. The strings must be separated by an arbitrary separator which can be specified in the `separator` attribute whose default is blank. The prevailing application is to successively bind variables to identifiers occurring in attributes of type IDREFS. It is surprising that none of the XML query languages proposed so far has such a capability. The last possibility to produce variable bindings by matching is to match a regular expression against a given text with the element `xql:string-match` . The regular expression is enhanced by variables that indicate which parts of the matched expressions ought to be bound to which variables.

YAXQL provides an element that allows to state a SQL query against a relational database server. This element, `xql:sql-query` , produces variable bindings. For every attribute in the result relation, a variable with the same name is bound to the according attribute value. For that, we require that the result attributes in the SQL query are named explicitly. This is the only restriction; otherwise arbitrary SQL queries are allowed.

A special element `xql:fill-form` allows to query Web sites with a form-based interface. User defined functions (UDFs) and user defined predicates (UDPs) can be declared using the `xql:fun-decl` and `xql:pred-decl` elements. The generic function and predicate call elements `xql:fun-call` and `xql:pred-call` allow their usage within YAXQL queries.

YAXQL has several built-in functions. Among these are set-operations, join-operations (especially outer joins), conditional expressions (`if` and `switch`), and simple tree manipulation functions for including/excluding parts of a document. The conditional expressions are useful not only in queries but also in result construction. The tree manipulation functions support several kinds of conditional exclusion of subtrees, restriction by indexed child ranges, and subtree replacement.

As (almost) common in query languages, YAXQL supports universal and existential quantifiers as well as grouping with an additional optional having.

A more unique feature of YAXQL is its ability to define form-based interfaces with attached queries. This part of XQL is the second possibility (besides `xql:evaluate`) to integrate queries into XML documents. This is a topic that has been neglected by prior XML query designers. Given an XQL document, the `xql:ask-for-variable-binding` element allows to specify a field that can be filled by the user. The user's input is then bound to a variable specified in the `xql:ask-for-variable-binding` element. Another attribute allows to specify a title for the form. All the variables bound explicitly or by filling in forms constitute the global environment for queries contained in the XML document. Query evaluation can be triggered by clicking on panels defined by `xql:eval-qc-on-click` which references a query-construct-pair.

7 Conclusion

We presented a list of classes of requirements and discussed the requirements contained in these classes. We further noted *en passant* that all the existing query languages for XML fail on almost every single requirement in this list. We then presented the XML query language YAXQL that meets the requirements. YAXQL exhibits such unique features as free variables with a semantics based on a 4-valued logic, full tree matching capabilities, full-text search facilities, an interface to relational databases, user defined functions and predicates. It allows not only to query values (as all other query languages) but also to query references to XML fragments. Reuse of queries and construct clauses is built-in. Last not least, YAXQL covers full XML. We further demonstrated how YAXQL queries can be embedded into documents, a necessity not mentioned in other proposals for XML query languages.

Acknowledgment The authors thank Peter Fankhauser and Carl-Christian Kanne for many fruitful discussions and comments on YAXQL.

A DTDs and Sample Documents

A.1 The EC-Server

DTD

```
<!ELEMENT ec:classes ((ec:class | ec:enzyme)*)>
<!ELEMENT ec:class (ec:link*)>
<!ATTLIST ec:class
          name  ID    #REQUIRED
          level CDATA #REQUIRED>
```

```
<!ELEMENT ec:link  (#PCDATA)>
<!ATTLIST ec:link
          xml:link  CDATA #FIXED "simple"
          href      CDATA #REQUIRED>

<!ELEMENT ec:enzyme (ec:name, ec:reaction, ec:comment*)>
<!ATTLIST ec:enzyme
          name ID #REQUIRED>
<!ELEMENT ec:name     (#PCDATA)>
<!ELEMENT ec:reaction (#PCDATA)>
<!ELEMENT ec:comment  (#PCDATA)>
```

Sample Document

```
<?xml version="1.0"?>
<!DOCTYPE ec:classes SYSTEM "ec.dtd">
<!-- This document contains the internal nodes of the EC-Classification -->

<ec:classes>
<ec:class name="-.-.-.-" level="0">
 <ec:link xml-link="SIMPLE" href="id(1.-.-.-)">1.-.-.- Oxidoreductase</ec:link>
 <ec:link xml-link="SIMPLE" href="id(2.-.-.-)">2.-.-.- Transferase</ec:link>
 <ec:link xml-link="SIMPLE" href="id(3.-.-.-)">3.-.-.- Hydrolase</ec:link>
 <ec:link xml-link="SIMPLE" href="id(4.-.-.-)">4.-.-.- Glycosidase</ec:link>
 <ec:link xml-link="SIMPLE" href="id(5.-.-.-)">5.-.-.- Lyase</ec:link>
 <ec:link xml-link="SIMPLE" href="id(6.-.-.-)">6.-.-.- Isomerase</ec:link>
</ec:class>

<ec:class name="1.-.-.-" level="1">
  <ec:link xml-links="SIMPLE href="id(1.1.-.-)">
    1.1.-.- Acting with the CH-OH group of donors
  </ec:link ...>
  ...
</ec:class>
...
<ec:enzyme name="1.1.4.1" level="4">
  <ec:name>Vitamin-K-epoxide reductase</ec:name>
  <ec:reaction>
    2-methyl-3-phytyl-1,4-naphtoquinone
    + oxidized dithiothreitol
    + H(2)O
    =
    2,3-epoxy-2,3-dihydro-2methyl-1,4-naphtoquinone
    + 1,4-dithiothreitol
  </ec:reaction>
  <ec:comment>
    In the reverse reaction, vitamin K 2,3-epoxide is reduced to
    vitamin K and possibly to vitamin K hydroquinone by
    1,4-dithioerythritol, wich is oxidized to a disulfide;
    some other dithiols and butane-4-thiol can also act.
  </ec:comment>
  <ec:comment>
    Inhibited strongly by warfarin (cf EC 1.1.4.2)
  </ec:comment>
</ec:enzyme>

...

</ec:classes>
```

A.2 The EDB-Server

DTD

```
<!ELEMENT edb:enzymes (edb:enzyme*)>
<!ELEMENT edb:enzyme (edb:ec-class, edb:sysname, edb:recname, edb:name*, edb:ph-opt,
                      edb:temp-opt, edb:inhibitor*)>
```

```
<!ELEMENT edb:ec-class  (#PCDATA)>
<!ELEMENT edb:sysname   (#PCDATA)>
<!ELEMENT edb:recname   (#PCDATA)>
<!ELEMENT edb:name      (#PCDATA)>
<!ELEMENT edb:ph-opt    (#PCDATA)>
<!ELEMENT edb:temp-opt  (#PCDATA)>
<!ELEMENT edb:inhibitor (#PCDATA)>
```

Sample Document

```
<?xml version="1.0"?>
<!DOCTYPE edb:enzymes SYSTEM "edb.dtd">
<edb:enzymes>
 <edb:enzyme>
  <edb:ec-class>1.1.4.1</edb:ec-class>
  <edb:sysname>1-Methyl-3-phytyl-1,4-naphthoquinone:oxidized-dithiothreitol
          oxidoreductase</edb:sysname>
  <edb:recname>Vitamin-K-epoxide reductase</edb:recname>
  <edb:name>phylloquinone expoxide reductase</edb:name>
  <edb:name>Vitamin K epoxide reductase</edb:name>
  <edb:name>Vitamin K1 epoxide reductase</edb:name>
  <edb:ph-opt>9.0</edb:ph-opt>
  <edb:temp-opt>25</edb:temp-opt>
  <edb:inhibitor>Warfarin</edb:inhibitor>
  <edb:inhibitor>Cholate</edb:inhibitor>
  <edb:inhibitor>Coumarin anticoagulants</edb:inhibitor>
  <edb:inhibitor>Deriphat 160</edb:inhibitor>
  <edb:inhibitor>DTT</edb:inhibitor>
  <edb:inhibitor>Lapachol</edb:inhibitor>
  <edb:inhibitor>Imidazopyridine</edb:inhibitor>
  <edb:inhibitor>Difenacoum</edb:inhibitor>
 </edb:enzyme>
</edb:enzymes>
```

A.3 The RDB-Server

DTD and Sample Document

```
<?xml version="1.0"?>
<!DOCTYPE rdb:pathways [
  <!ELEMENT rdb:pathways (rdb:substrate*, rdb:reaction*, rdb:pathway*)>
  <!ELEMENT rdb:substrate EMPTY>
  <!ATTLIST rdb:substrate
            sid  ID    #REQUIRED
            name CDATA #REQUIRED>
  <!ELEMENT rdb:reaction EMPTY>
  <!ATTLIST rdb:reaction
            name   ID     #REQUIRED
            input  IDREFS #REQUIRED
            enzyme CDATA  #REQUIRED
            output IDREFS #REQUIRED>
  <!ELEMENT rdb:pathway EMPTY>
  <!ATTLIST rdb:pathway
            name      ID     #REQUIRED
            reactions IDREFS #REQUIRED>

]>

<rdb:pathways>
<rdb:substrate sid="s1" name="2-methyl-3-phytyl-1,4-naphtoquinone"/>
<rdb:substrate sid="s2" name="oxidized dithiothreitol"/>
<rdb:substrate sid="s3" name="H(2)0"/>
<rdb:substrate sid="s4" name="2,3-epoxy-2,3-dihydro-2methyl-1,4-naphtoquinone"/>
<rdb:substrate sid="s5" name="1,4-dithiothreitol"/>

<rdb:reaction name="r1"
              input="s1"
```

```
            enzyme="Vitamin-K-epoxide reductase"
            output="s2 s3"/>

<rdb:pathway name="p1" reactions="r1 r2 r3 r4 r5"/>

</rdb:pathways>
```

B DTD for YAXQL

```
<?xml version="1.0"?>

<!-- XQL DTD -->

<!-- Attribute groups -->

<!ENTITY % XQL:SRC ' href   CDATA #IMPLIED
                     select CDATA #IMPLIED'>

<!ENTITY % XQL:DATA-TYPE ' data-type  CDATA  #IMPLIED'>

<!ENTITY % XQL:VAR       'var CDATA  #REQUIRED'>
<!ENTITY % XQL:TVAR      ' %XQL:VAR; %XQL:DATA-TYPE;'>
<!ENTITY % XQL:NULL      ' maybe-null  (YES|NO) "NO"'>
<!ENTITY % XQL:NTVAR     ' %XQL:TVAR; %XQL:NULL; '>
<!ENTITY % XQL:UNBOUND   ' maybe-unbound (YES|NO) "NO" '>
<!ENTITY % XQL:SRC-NTVAR ' %XQL:SRC; %XQL:NTVAR;'>

<!ENTITY % XQL:CO 'cardinality-only  (YES|NO)    "NO"
                   card-var           CDATA      #IMPLIED'>

<!ENTITY % XQL:DIST    'distinct (YES|NO) "YES" '>
<!ENTITY % XQL:BIND-TO 'bind-to   CDATA   #IMPLIED '>
<!ENTITY % XQL:ORDERED 'ordered  (YES|NO) "NO" '>
<!ENTITY % XQL:ORDER-BY 'order-by CDATA #IMPLIED'>

<!ENTITY % XQL:FT 'from CDATA #REQUIRED
                   to   CDATA #REQUIRED'>

<!ENTITY % XQL:PATTERN-ATTRS '%XQL:NTVAR; %XQL:BIND-TO; %XQL:ORDERED;
        deep-match     (YES|NO) "NO"
        preserve-order (YES|NO) "NO"
        optional-match (YES|NO) "NO"'>

<!ENTITY % XQL:REGEXPR 'regexpr CDATA #IMPLIED'>

<!-- All Entities -->

<!ENTITY % XQL:CMP '(xql:eq | xql:neq | xql:geq | xql:leq | xql:less | xql:greater |
                     xql:is-bound | xql:is-unbound | xql:is-null | xql:is-not-null |
                     xql:contains | xql:pred-call)'>

<!ENTITY % XQL:BCON  '(xql:implies | xql:and | xql:or | xql:not)'>
<!ENTITY % XQL:QUANT '(xql:for-all | xql:exists)'>
<!ENTITY % XQL:BXP   '(%XQL:CMP; | %XQL:BCON; | %XQL:QUANT;)'>

<!ENTITY % XQL:SET-OP  '(xql:union | xql:intersect | xql:set-minus)'>
<!ENTITY % XQL:ALG-OP  '(xql:join  | (%XQL:SET-OP;))'>
<!ENTITY % XQL:QUERY   '(xql:query | xql:query-reference |
                         xql:recursive-query | (%XQL:ALG-OP;))'>
<!ENTITY % XQL:MATCHER '(xql:match | xql:string-match | xql:iterate)'>
<!ENTITY % XQL:BINDER  '(xql:bind | xql:bindings-list | xql:sql-query |
                          (%XQL:QUERY;) | (%XQL:MATCHER;))'>
<!ENTITY % XQL:PRED    '(%XQL:BXP;)'>
<!ENTITY % XQL:MATCH-EXPR  '(xql:match | xql:string-match)'>

<!ENTITY % XQL:AGGR '(xql:min | xql:max | xql:count | xql:sum | xql:avg)'>
```

```
<!ENTITY % XQL:VAR-STMT '(xql:declare | xql:bind | xql:unbind)'>

<!ENTITY % XQL:EXPR '(xql:value | xql:deref | xql:value-of | xql:get-tag | xql:get-attribute |
                     xql:source | xql:element | xql:attribute | xql:add | xql:substract |
                     xql:multiply | xql:divide | xql:modulo | xql:conc | xql:range |
                     xql:fun-call | %XQL:AGGR; | %XQL:PRED;)'>

<!ENTITY % XQL:STMT '(%XQL:EXPR; | %XQL:VAR-STMT; | xql:if | xql:switch | xql:for |
                     xql:for-each-match | xql:evaluate)'>

<!-- XQL top level elements -->

<!ENTITY % XQL:XQL-TOP '(%XQL:QUERY; | %XQL:MATCHER; |
                        xql:query-construct | xql:query-construct-reference |
                        xql:sql-query | %XQL:STMT;)'>

<!ELEMENT xql:stmts ((%XQL:XQL-TOP;)*)>
<!ELEMENT xql:stmt   (%XQL:XQL-TOP;)>

<!-- Very simple query  -->

<!ELEMENT xql:for-each-match ANY>
<!ATTLIST xql:for-each-match %XQL:SRC-NTVAR; %XQL:DIST; %XQL:ORDER-BY;
          preserve-order   (YES|NO) #IMPLIED>

<!-- Query Construct Elements -->

<!ELEMENT xql:query-construct
          ((%XQL:QUERY;)*, (xql:construct | xql:construct-reference))>
<!ATTLIST xql:query-construct name ID #REQUIRED>

<!ELEMENT xql:query-construct-reference EMPTY>
<!ATTLIST xql:query-construct-reference  href CDATA #REQUIRED>

<!ELEMENT xql:query (xql:declare*, (%XQL:BINDER;)*, (%XQL:PRED;)?,
                     (xql:group-by, (xql:having)?)?, xql:project?)>
<!ATTLIST xql:query name ID #REQUIRED>

<!ELEMENT xql:query-reference (xql:bind*, (%XQL:PRED;)?, xql:project?)>
<!ATTLIST xql:query-reference href  CDATA  #REQUIRED>

<!ELEMENT xql:group-by (%XQL:EXPR;)*>
<!ELEMENT xql:having   (%XQL:PRED;)>

<!ELEMENT xql:project (xql:bind*)>
<!ATTLIST xql:project  %XQL:CO; %XQL:DIST; %XQL:ORDER-BY;
          group-by           CDATA      #IMPLIED>

<!ELEMENT xql:construct ANY>
<!ATTLIST xql:construct out-dtd CDATA #IMPLIED>

<!ELEMENT xql:construct-reference EMPTY>
<!ATTLIST xql:construct-reference href CDATA #REQUIRED>

<!ELEMENT xql:evaluate (((xql:bind | xql:unbind)*,
                         (%XQL:QUERY;, %XQL:PRED;?, xql:project?),
                         ((xql:construct | xql:construct-reference))) |
                         xql:query-construct-reference)>

<!ATTLIST xql:evaluate %XQL:CO; %XQL:ORDER-BY;
          name           ID      #REQUIRED
          top-element    CDATA   #IMPLIED
          bottom-element CDATA   #IMPLIED>

<!ELEMENT xql:join ((%XQL:QUERY;), (%XQL:QUERY;), (%XQL:PRED;))>
<!ATTLIST xql:join outer (NO|LEFT|RIGHT|FULL) "NO">
```

```
<!ELEMENT xql:union     ((%XQL:QUERY;)*)>
<!ELEMENT xql:intersect ((%XQL:QUERY;)*)>
<!ELEMENT xql:set-minus ((%XQL:QUERY;)*)>

<!-- Recursive queries -->

<!ELEMENT xql:recursive-query (xql:declare*, xql:union, (%XQL:PRED;)?, xql:project)>
<!ATTLIST xql:recursive-query name ID #REQUIRED>

<!-- embedding SQL queries -->
<!ELEMENT xql:sql-query (xql:declare*, xql:xql-text)>
<!ELEMENT xql:sql-text  (#PCDATA)>
<!ATTLIST xql:sql-query name     ID      #REQUIRED
                        server   CDATA   #REQUIRED
                        database CDATA   #REQUIRED>


<!-- Grouping in construct clauses -->

<!ELEMENT xql:for-each ANY>
<!ATTLIST xql:for-each vars CDATA #REQUIRED %XQL:ORDER-BY;>

<!-- Explicit grouped bindings -->

<!ELEMENT xql:bindings-list (xql:bindings*)>
<!ELEMENT xql:bindings (xql:bind)*>

<!-- Binding by tree matching -->

<!ELEMENT xql:match ((%XQL:MATCH-EXPR;)*)>
<!ATTLIST xql:match %XQL:SRC; %XQL:PATTERN-ATTRS;
          max-gap         CDATA  #IMPLIED
          max-level-diff  CDATA  #IMPLIED
          near            CDATA  #IMPLIED>

<!-- Binding by string matching -->

<!ELEMENT xql:string-match EMPTY>
<!ATTLIST xql:string-match %XQL:SRC; %XQL:REGEXPR;>

<!-- Binding by iteration through lists -->

<!ELEMENT xql:iterate EMPTY>
<!ATTLIST xql:iterate %XQL:TVAR; %XQL:SRC; %XQL:REGEXPR;
          separator  CDATA  #IMPLIED>

<!-- Variable declaration, binding, dereferenciation etc. -->

<!ELEMENT xql:declare EMPTY>
<!ATTLIST xql:declare %XQL:NTVAR; %XQL:UNBOUND;
          default        CDATA       #IMPLIED>

<!ELEMENT xql:bind  ANY>
<!ATTLIST xql:bind  %XQL:NTVAR; %XQL:UNBOUND; %XQL:SRC; %XQL:BIND-TO;>

<!ELEMENT xql:value EMPTY>
<!ATTLIST xql:value %XQL:VAR;>

<!ELEMENT xql:deref EMPTY>
<!ATTLIST xql:deref var CDATA #REQUIRED>

<!ELEMENT xql:unbind EMPTY>
<!ATTLIST xql:unbind var CDATA #REQUIRED>

<!ELEMENT xql:constant ANY>
<!ATTLIST xql:constant  %XQL:SRC; %XQL:BIND-TO;
```

```
          const          CDATA       #REQUIRED
          data-type      CDATA       #IMPLIED>

<!-- Boolean expressions -->

<!ELEMENT xql:eq      ((%XQL:EXPR;)+)> <!ELEMENT xql:neq     ((%XQL:EXPR;)+)>
<!ELEMENT xql:geq     ((%XQL:EXPR;)+)> <!ELEMENT xql:leq     ((%XQL:EXPR;)+)>
<!ELEMENT xql:less    ((%XQL:EXPR;)+)> <!ELEMENT xql:greater ((%XQL:EXPR;)+)>

<!ATTLIST xql:eq      %XQL:DATA-TYPE;> <!ATTLIST xql:neq     %XQL:DATA-TYPE;>
<!ATTLIST xql:geq     %XQL:DATA-TYPE;> <!ATTLIST xql:leq     %XQL:DATA-TYPE;>
<!ATTLIST xql:less    %XQL:DATA-TYPE;> <!ATTLIST xql:greater %XQL:DATA-TYPE;>

<!ELEMENT xql:is-bound EMPTY>    <!ATTLIST xql:is-bound var CDATA #REQUIRED>
<!ELEMENT xql:is-unbound EMPTY>  <!ATTLIST xql:is-unbound var CDATA #REQUIRED>

<!ELEMENT xql:is-null EMPTY>
<!ATTLIST xql:is-null var CDATA #REQUIRED>
<!ELEMENT xql:is-not-null EMPTY>
<!ATTLIST xql:is-not-null var CDATA #REQUIRED>

<!ELEMENT xql:contains ANY>
<!ATTLIST xql:contains %XQL:SRC; %XQL:REGEXPR;>

<!ELEMENT xql:implies ((%XQL:BXP;), (%XQL:BXP;))>
<!ELEMENT xql:and     ((%XQL:BXP;)*)>
<!ELEMENT xql:or      ((%XQL:BXP;)*)>
<!ELEMENT xql:not     ((%XQL:BXP;))>

<!ELEMENT xql:for-all ((%XQL:QUERY;), (%XQL:PRED;))>
<!ELEMENT xql:exists  ((%XQL:QUERY;), (%XQL:PRED;))>

<!-- Expressions -->

<!ELEMENT xql:source EMPTY>
<!ATTLIST xql:source %XQL:SRC; >

<!ELEMENT xql:attach ANY>
<!ATTLIST xql:attach %XQL:SRC;>

<!ELEMENT xql:replace ANY>
<!ATTLIST xql:replace %XQL:SRC;>

<!-- Todo: arguments for the following two beasts -->
<!ELEMENT xql:exclude-if ((%XQL:PRED;) | (%XQL:MATCH-EXPR;))>
<!ATTLIST xql:exclude-if %XQL:SRC;>
<!ELEMENT xql:exclude-if-not ((%XQL:PRED;) | (%XQL:MATCH-EXPR;))>
<!ATTLIST xql:exclude-if-not %XQL:SRC;>

<!ELEMENT xql:conc ((%XQL:EXPR;)*)>
<!ATTLIST xql:conc separator CDATA #IMPLIED>

<!ELEMENT xql:fun-call ((%XQL:EXPR;)*)>
<!ATTLIST xql:fun-call
          fun-name   CDATA    #REQUIRED
          fanced    (YES|NO)  "YES">

<!ELEMENT xql:pred-call ((%XQL:EXPR;)*)>
<!ATTLIST xql:pred-call
          pred-name   CDATA    #REQUIRED
          fanced     (YES|NO)  "YES">

<!ELEMENT xql:range (%XQL:EXPR;)>
<!ATTLIST xql:range %XQL:SRC; %XQL:FT;>

<!ELEMENT xql:if ((%XQL:PRED;), (%XQL:EXPR;), (%XQL:EXPR;)?)>
<!ELEMENT xql:switch ((%XQL:EXPR;), xql:case*)>
```

```
<!ELEMENT xql:case ((%XQL:STMT;)*)>
<!ELEMENT xql:for  ((%XQL:STMT;)*)>
<!ATTLIST xql:for var    CDATA #REQUIRED
                  start  CDATA #REQUIRED
                  step   CDATA #REQUIRED
                  stop   CDATA #REQUIRED>

<!ELEMENT xql:min   (%XQL:EXPR;)>  <!ELEMENT xql:max   (%XQL:EXPR;)>
<!ELEMENT xql:count (%XQL:EXPR;)>  <!ELEMENT xql:sum   (%XQL:EXPR;)>
<!ELEMENT xql:avg   (%XQL:EXPR;)>

<!-- Function declaration (UDFs and UDPs) -->

<!ENTITY % XQL:FUN-ATTRS ' fun-name  CDATA  #REQUIRED
                           language  CDATA  #REQUIRED
                           source    CDATA  #REQUIRED
                           binary    CDATA  #REQUIRED'>

<!ELEMENT xql:fun-arg EMPTY>
<!ATTLIST xql:fun-arg arg-name    CDATA #REQUIRED
                      data-type   CDATA #REQUIRED>

<!ELEMENT xql:fun-decl  (#PCDATA)>
<!ATTLIST xql:fun-decl %XQL:FUN-ATTRS;
          return-type     CDATA         #REQUIRED>

<!ELEMENT xql:pred-decl (#PCDATA)>
<!ATTLIST xql:pred-decl %XQL:FUN-ATTRS;>

<!-- Form filling -->

<!ELEMENT xql:fill-form (xql:form-entry)*>
<!ATTLIST xql:fill-form  %XQL:SRC; var CDATA #REQUIRED >

<!ELEMENT xql:form-entry ANY>
<!ATTLIST xql:form-entry name CDATA #REQUIRED>

<!-- User interaction -->
<!ELEMENT xql:ask-for-variable-binding EMPTY>
<!ATTLIST xql:ask-for-variable-binding %XQL:NTVAR;
          title      CDATA  #REQUIRED>

<!ELEMENT xql:eval-qc-on-click EMPTY>
<!ATTLIST xql:eval-qc-on-click name  IDREF   #REQUIRED
                               href  CDATA   #IMPLIED
                               title CDATA   #REQUIRED>

<!-- END OF XQL DTD -->
```

References

[1] S. Abiteboul. On Views and XML. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 1–9, 1999.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1):5–19, April 1997.

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

[4] G. Arocena and A. Mendelzon. WebOQL: restructuring documents, databases and webs. In *Proc. IEEE Conference on Data Engineering*, pages 24–33, 1998.

[5] A. Bosworth, A. Levy, J. Widom, R. Goldman, J. McHugh, A. Layman, A. Ardelwanu, and D. Schach. Position paper for the W3C query language workshop december 3, 1998. In *[26]*, 1998.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation 10-Feb-98.

[7] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 505–516, 1996.

[8] J. Clark. XSL transformations (XSLT) version 1.0. Technical report, World Wide Web Consortium, 1999. W3C Recommendation 16 Nov. 1999.

[9] J. Clark and S. DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium, 1999. W3C Recommendation 16 Nov. 1999.

[10] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 177–188, Seattle, WA, 1998.

[11] S. Cluet, S. Jacqumin, and J. Simeon. The new YATL. Technical report, INRIA, 1999.

[12] P. Cotton and A. Malhotra. Candidate requirements for xml query. In *[26]*, 1998.

[13] S. Deach. Extensible stylesheet language (XSL) specification. Technical report, World Wide Web Consortium, 1999. W3C Working Draft 21 Apr 1999.

[14] S. DeRose. XQuery: a unified syntax for linking and querying general XML documents. In *[26]*, 1998.

[15] S. DeRose, C. Sperberg-McQueey, and B. Smith. Queries on links and hierarchies. In *[26]*, 1998.

[16] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: a query language for XML. Technical report, World Wide Web Consortium, 1989. http://www.w3.org/TR/NOTE-xml-ql.

[17] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 414–425, 1998.

[18] M. Fernandez, J. Simeon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widon. XML query languages: Expericences and exemplars. http://www-db.research.bell-labs.com/user/simeon/xquery.html, 1999.

[19] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.

[20] R. Himmeröder, G. Lausen, B. Ludäscher, and C. Schlepphorst. Querying the web with FLORID. In *Proc. der GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft (BTW)*, pages 47–51, 1998.

[21] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, 1992.

[22] D. Konopnicki and O. Shmueli. W3QS: a query system for the world-wide web. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 54–65, Dublin, Ireland, 1995.

[23] L. Lakshmanan, F. Sadri, and V. Subramanian. A declarative language for querying and restructuring the web. In *Int. Workshop on Research Issues in Data Engineering (RIDE)*, pages 12–21, 1996.

[24] D. Maier. Database desiderata for an XML query language. In *[26]*, 1998.

[25] E. Maler and S. DeRose. Xml pointer language (xpointer). Technical report, World Wide Web Consortium, 1998. World Wide Web Consortium Working Draft 03-March-1998.

[26] M. Marchiori. Ql'98 - the query languages workshop. Technical report, W3C, Dec. 1998. http://www.w3.org/TandS/QL/QL98.

[27] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, pages 80–91, 1996.

[28] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, April 1997.

[29] D. Quass. Ten features necessary for an XML query language. In *[26]*, 1998.

[30] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *[26]*, 1998.