Ten Features Necessary for an XML Query Language

Dallan Quass
Brigham Young University
quass@byu.edu

The purpose of this article is not to put forth another query language proposal for XML, but to outline ten features that any XML query language proposal should include. It is hoped that presenting such features will give the reader a better way to judge the query language proposals in existence. Each feature is described below, along with a short example. The purpose of the examples is solely for clarification; the syntax used in the examples is not important.

1. Clean Semantics

Any query language for XML must be able to express simple queries simply. A good XML query language should be usable by novice web-users, not just database experts. They will use the XML query language primarily for filtering and restructuring XML data, so these queries should be easy to express. More complex queries involving say, aggregation or universal quantification, should be expressible as well but may involve more complexity.

One possibility for a clean semantics is to base the query language upon the SQL *select-from-where* statement.

> SELECT *expr*
> FROM *path*
> WHERE *cond*

The semantics of such a statement can be interpreted as "for each path *path* found in the XML data where condition *cond* holds on the path, produce an element in the query result defined by *expr*. If multiple paths were contained in the FROM clause, the semantics would change to "for each combination of paths *path1* .. *pathN* found in the XML data..." similar to multiple table references in SQL. For example, the following query selects employee names and phone numbers from an XML document.

Example Source document

```
<employees>
<emp>
<name>John Taylor</name>
<age>34</age>
<phone type=home>555-1234</phone>
<address>
<street>123 N. Main</street>
<city>Chicago</city>
<state>IL</state>
</address>
</emp>
<emp>
<name>Sue Smith</name>
```

```
<age>31</age>
<phone type=home>555-9999</phone>
<phone type=fax>555-8887</phone>
<address>
<street>100 W. State</street>
<city>Salt Lake City</city>
<state>UT</state>
</address>
</emp>
</employees>
```

Query

```
SELECT <newemp> e.name e.phone </newemp>
FROM employees.emp e
```

Query Result

```
<newemp>
<name>John Taylor</name>
<phone type=home>555-1234</phone>
</newemp>
<newemp>
<name>Sue Smith</name>
<phone type=home>555-9999</phone>
<phone type=fax>555-8887</fax>
</newemp>
```

2. Path Expressions

Since XML elements often contain child elements, the query language should support "path expressions," which allow the writer easy access to nested elements. For example, the query

```
SELECT <newemp> e.name e.address.city </newemp>
FROM employees.emp e
```

returns the name and city of each employee.

3. Ability to Return an XML Document

The standard behavior of most query languages is to return a set of elements of some type. For an XML query language the returned value should be an XML document. One possibility for returning an XML document is to allow the query return value to be embedded within XML markup tags. For example, the following query returns its results embedded within <result> </result> tags. The <result> </result> tags could be applied implicitly if not explicitly associated with the query.

Query

```
<result>
SELECT <newemp> e.name </newemp>
FROM employees.emp e
```

```
</result>
```

Query Result

```
<result>
<newemp>
<name>John Taylor</name>
</newemp>
<newemp>
<name>Sue Smith</name>
</newemp>
</result>
```

4. Ability to Query and Return XML Tags and Attributes

An XML data element contains data, a tag, and optionally, attributes. It is imperitave that an XML query language be able to query the element tags and attributes as well as the data. Tags and attributes should be able to be referenced in any part of the query--the SELECT clause, the FROM clause, and the WHERE clause. For example, the following query selects all employees with a fax number and returns their name as an attribute of the <newemp> element.

Query

```
SELECT <newemp name= e.name.value/>
FROM employees.emp e
WHERE e.phone.type = "fax"
```

Query Result

```
<newemp name= "Sue Smith"/>
```

Similarly, the following query "inverts" the employees’ names and emp tags..

Query

```
SELECT <e.name> <recordtag>e.tag</recordtag> </e.name>
FROM employees.emp e
```

Query Result

```
<John Taylor>
<recordtag>emp</recordtag>
</John Taylor>
<Sue Smith>
<recordtag>emp</recordtag>
</Sue Smith>
```

5. Intelligent Type Coercion

Since both textual and numeric data are represented as strings in XML, the query language should be intelligent enough in comparison operations to determine whether a string comparison is intended or if a coercion is required. For example, the following query requests all employees whose age is less than 100. Even though "34" > "100" using a string comparison operator, it is clear from the query (since 100 is written as a number) that the intent is to use a numeric comparison.

```
SELECT e
FROM employees.emp e
WHERE e.age < 100
```

The following query, however, suggests using string comparison, since 100 is written as a string.

```
SELECT e
FROM employees.emp e
WHERE e.age < "100"
```

6. Handle Unexpected Data

Data in XML does not have to conform to a fully-structured DTD. Therefore, it is critical that an XML query language "do the right thing" in the face of unexpected data as much as possible. For example, suppose that a query writer expected only a single occurrence of the phone element for each employee, and so wrote the following query to return the employees whose phone number was "555-1234."

```
SELECT e
FROM employees.emp e
WHERE e.phone = "555-1234"
```

In order to handle employee elements having multiple phone elements (or no phone elements), the above query should interpret the WHERE clause condition with an implicit existential quantification. That is, "there exists an e.phone element *x* such that *x* = "555-1234."

7. Allow Queries When the DTD Is Not Fully Known

It may often be the case with XML that the query writer understands a part of the DTD, but not in its entirety. The query language needs to support wildcards in the path expressions to allow the query writer to "skip past" parts of the document structure of which he or she is not aware. For example, suppose that the query writer was uncertain as to the element tag for phone number (but knew it was either phone or number) and the exact path leading to employee city. The following query allows the writer to return phone numbers for all employees containing an element "Chicago." In the query a "*" represents any sequence of 0 or more elements.

```
SELECT <newemp> e.(phone|number) </newemp>
FROM employees.emp e
WHERE e.* = "Chicago"
```

8. Return Unnamed Attributes

In restructuring information it is often useful to express things like "return all child elements," or "return all child elements except this one." An XML query language should support queries that return elements even when their tags are unknown. For example, the following query returns all employee child elements except phone. In the query, a "^" is used similarly as in regular expression character classes to negate the set.

```
SELECT <newemp> e.(^ phone) </newemp>
FROM employees.emp e
```

9. Return Trees Instead of Sets

Although query languages usually return sets, when restructuring an XML document it is likely that the user will want to have the query return a more nested structure. For example, suppose in our example database the query writer wanted to reorganize the data so that employees were grouped by city within state. The general idea would be to have a hierarchy of tags with cities embedded within state tags, and employees embedded within city tags as in the following.

```
<state>
<statename>..</statename>
<city>
<cityname>..</cityname>
<emp>...</emp>
</city>
</state>
```

One possibility for doing this would be to add a new construct, say "merge by," similar to group by in SQL, that would take the elements returned from a select-from-where query and merge them according to attributes. For example, the above structure might be specified as:

```
SELECT <state> <statename> e.address.state </statename> <city> <cityname>
e.address.cityname </cityname> e </city> </state>
FROM employees.emp e
MERGE state by statename, city by cityname
```

10. Preserve Order

SGML documents have an implicit order, and XML documents do as well. It is important for an XML query language to be able to optionally guarantee that the order of returned results is the same as in the original document. Perhaps an extension to the "order by" clause could be used for this purpose. For example,

```
SELECT e
FROM employees.emp e
ORDER BY document-order
```