Designing A Language for Querying XML Data

Hiroshi Ishikawa, [+] Kazumi Kubota, [++] and Yasuhiko Kanemasa [++]

XML data have begun to be widely used in EC/EDI applications, digital libraries, and Web information systems. Such applications usually use a large number of XML data. First, we must allow users to retrieve only necessary portions of XML data by specifying search conditions to flexibly describe such applications. Second, we must allow users to combine XML data from different sources to produce new XML data. To this end, we will provide a query language for XML data called *XQL*. We have designed XQL, keeping in mind its continuity with database standards such as SQL and OQL although we don't stick to its strict conformity. In this paper, we describe the requirements for a query language for XML data and explain the functionality of XQL with its semantics based on set theory. Finally, we make brief comments on the implementation of XQL.

1. Introduction

XML data are expected to be widely used in Web information systems and EC/EDI applications. Such applications usually use a large number of XML data. First, we must allow users to retrieve only necessary portions of XML data by specifying search conditions to flexibly describe such applications. Second, we must allow users to combine XML data from different sources. To this end, we will provide a query language for XML data tentatively called *XQL* (Xml data Query Language) [6], which has coincidentally the same name as a query language proposed by Microsoft and et al.[8]

One goal of XQL is to provide integrated access to heterogeneous data sources with different schemas. For example, XQL enables the client to retrieve the cheapest shopper for the same item from multiple on-line shopping sites like so-called *infomediary*[4] or *information intermediation* businesses on the Internet (See Figure 1) although catalogues and prices may have different structures depending on shops, which we call *semi-structured* data. Another goal associated with heterogeneity is to combine heterogeneous data-intensive applications. For example, XQL enables the client to check where the ordered item exists by combining the order-entry system and the item tracking system with the same order number. Thus, XQL will serve as a central role for constructing both kinds of EC applications: Business to Customer and Business to Business.

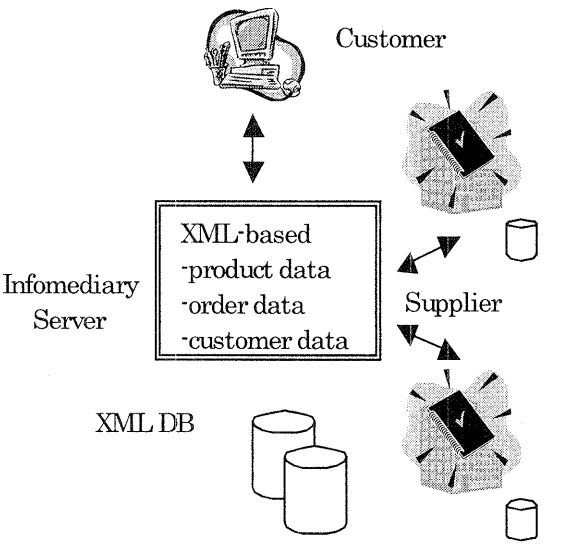

Figure 1. An XML data-intensive application

A second goal of XQL is to select digital contents by specifying search conditions on XML-based descriptions. For example, XQL will help the customer or mediator to find digital contents satisfying search conditions on XML-based properties, distributed at different sites such as digital libraries and museums. XQL will also help to find interesting programs among hundreds of digital TV channels by querying XML-based EPG (Electronic Program Guide).

A third goal of XML is to tailor XML data published at Web sites for custom needs. For example, XQL helps filter technical manuals containing different versions to produce a manual for a dedicated purpose. XQL also allows

[+] Tokyo Metropolitan University
[++] Fujitsu Laboratories Ltd.

the customer to gather only necessary information from disclosure information published at EDGAR-like sites for making customized reports.

A query language called XML-QL [3] has already been proposed to W3C, which has largely inspired the design of XQL. The first workshop on XML query language was successfully held [9]. As a result, W3C has launched a workgroup *XML-Query* to propose standard query languages for XML data. We have designed XQL, keeping in mind its continuity with standards such as SQL [1] and OQL [2] although we don't stick to its strict conformity. We will describe the requirements for a query language for XML data in Section 2 and explain the functionality of XQL with its semantics based on set theory by using the example data in Section 3. We will compare XQL with other proposals and describe the current implementation of our XQL processing system in Section 4.

2. Requirements

We consider the following requirements as mandatory for a query language for XML data:

(1) XML query languages must take XML data as input and give XML as output. This is necessary to combine queries in succession. For example, the customers in the above infomediary services interactively refine search results of product data.

(2) XML query languages must understand features of XML data structures such as elements, tags, and attributes. In particular, the users must be able to specify hierarchical structures (i.e., nested structures) of tag paths in a query. This is a basic necessity for a data model of XML query languages, capturing semantics of applications such as infomediary services.

(3) XML query languages must provide operations on XML data, that is, XML versions of relational operators such as selection, join, sort, and grouping. XML query languages must be also able to produce new elements by using retrieved elements. This is a basic necessity for XML query languages themselves, enabling efficient application development such as infomediary services.

(4) XML query languages must view XML data as ordered sets of elements and must preserve the order. They must provide indexed access to ordered elements. For example, the customer requests recommendations from multiple suppliers and chooses the first one from each group of recommendations by using an index number. Further, they must provide set operators over XML data such as union, intersection. For example, infomediary services merge catalogue data from different suppliers by using a set operator union.

(5) XML query languages must be able to combine heterogeneous XML data from different sources specified by different URIs. For example, infomediary services combine ordering databases at suppliers and tracking databases at logistics with the same order number. They must be also able to provide universal access to multiple data sources even with slightly different schemas. For example, infomediary services merge price and availability data for the same item from different suppliers, which autonomously have slightly different schemas depending on suppliers. This is necessary for integrating Web-based XML data-intensive applications. Note that SQL and OQL don't provide this function.

(6) XML query languages must allow specification of regular expressions both on tag paths and text strings although full capability may be unnecessary because of its computational complexity. This is necessary to resolve semi-structured-ness of XML. For example, customer databases slightly differ from channel to channel such as the Internet, telephony, and postage for optimal design. Infomediary services combine such semi-structured databases to mine prospective customers. This is helpful for querying heterogeneous data sources and uncertain data sources. This is also helpful for flexibly filtering XML data. Note that neither SQL nor OQL provides this function.

(7) XML query languages must allow the users to define views on base XML data, which are analogous to relational views. That is, the users must be able to define XML data views as functions by using XML query languages and to specify such functions in a query.

This is necessary for tailoring base XML data for custom needs and reusing them. For example, frequently asked queries in infomediary services are defined as functions for reuse.

(8) An XML query must be embedded in XML data and be evaluated to produce XML data. This is necessary for extending dynamic aspects of XML data. For example, the prices embedded in product catalogues in infomediary services change in time course, so on-demand access through an embedded query is necessary for refreshing most current prices.

(9) XML query languages must keep syntactic and semantic continuity with other standards such as SQL and OQL although SQL and OQL's lack of the above important functions (5) and (6) necessitates new XML query languages other than SQL and OQL. This feature is necessary for increasing efficiency in application development such as infomediary services because the developers have already been familiar with the standards, which are nonprocedural.

(10) XML query languages must be processed efficiently. Efficiency is always a key to success in new applications such as infomediary services. The language processor must provide query optimization. The processor must use schema information if available although it doesn't assume the existence of schemas.

3. Design

3.1 Database schema

We use database schemas or DTD by slightly changing example DTD used in XML-QL[3] as follows:

```
<!ELEMENT bib (book*, article*)>
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author+, title, publisher)>
<!ATTLIST article year CDATA >
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname,
office+)>
<!ELEMENT office (#PCDATA | (building,
room))>
<!ELEMENT title (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT building (#PCDATA)>
<!ELEMENT room (#PCDATA)>
```

Here DTD for book elements indicates that a book has at least one author and one mandatory title and publisher. DTD for article elements indicates that an article has one mandatory author and title and one optional year. An article has a year as an attribute. Note that an author has at least one office, which has a variant structure of either literal data or a pair of building and room. Like this example, XML data are inherently semi-structured, which XML query language design must take into account.

The following is a part of XML data with conformity to the above DTD:

```
<bib>
  <book year="1993">
    <author>
      <firstname>Hiroshi</firstname>
      <lastname>Ishikawa </lastname>
      <office>
        <building> L2 </building>
        <room> S210 </room>
      </office>
    </author>
    <title>Object-Oriented DatabaseSystem
    </title>
    <publisher> Springer Verlag </publisher>
  </book>
  <book year="1996">
    <author>
      <firstname>Jeff </firstname>
      <lastname>Ullman </lastname>
      <office> Gates Building</office>
    </author>
    <author>
      <firstname>Jennifer</firstname>
      <lastname>Widom</lastname>
    </author>
    <title>A First Course in Database Systems
    </title>
    <publisher> Prentice-Hall</publisher>
  </book>
</bib>
```

We take an ordered directed graph as a logical model for an XML query language. That is, the data model of an XML query language can be represented as data structures consisting of nodes and directed edges, which are ordered.

3.2 Functionality

We describe the functionality of XQL by using schemas introduced in the previous section. XQL has a select-from-where construct as a basis, similar to SQL and OQL.

(1) Data match for selection

The basic function of XQL is to select arbitrary elements from XML data by specifying search conditions. The following query retrieves authors of books published before 1995 by Springer Verlag:

```
select $book.author
from bib URI "www.a.b.c/bib.xml",
     book $bib.book
where $book.publisher.name
     ="Springer Verlag" and
      $book.@year < "1995"
```

The basic unit of XQL is a path expression, that is, an *element variable* (explained just below) followed by a series of tag names such as "$bib.book". The user must declare at least one element variable in a from-clause such as bib and book. In particular, the user can bind XML data as input specified by URI to element variables such as bib. This declares a context where an XQL query is evaluated. References of element variables are done by prefixing "$" to them such as $bib.book. In a select-clause, the user specifies XML data as a result such as $book.author. Result elements have a tag for author as default in this case. The user checks a data match condition for selection in a where-clause. In general, conditions consist of logical combination of simple conditions such as $book.publisher.name ="Addison-Wesley". Note that attributes such as year are referenced similar to tags by prefixing "@" to attributes, such as $book.@year. Basically, two values of elements are compared in an alphabetical order.

The result of an XQL query is XML data. In our current design, the resultant XML data have no DTD, that is, they are well-formed XML data. As XQL doesn't assume the existence of DTD or schemas and it can retrieve just well-formed XML data, this causes no problem. However, we have a future plan to provide an option to dynamically create resultant DTD for valid XML with assistance of users' hints such as XML data construction specification in XQL queries.

For example, the result of the above query has the following structure, automatically wrapped by a tag "XQL:result":

```
<XQL:result>
  <author>
    <firstname>Hiroshi</firstname>
    <lastname>Ishikawa </lastname>
    <office>
      <building> L2 </building>
      <room> S210 </room>
    </office>
  </author>
  <author> ... </author>
  ...
</XQL:result>
```

In general, of course, the users have to know schemas for XML data before they submit queries. They usually consult DTD or schemas defined by currently proposed XML schema definition languages if there are any DTD or schemas available. Otherwise, a possible approach is to automatically extract schemas from individual XML data. However, we currently think that this is not a mature technology applicable to XQL.

We briefly describe the semantics of XQL based on set theory. A set of XML elements is either ordered or unordered. XML elements have hierarchical structures; XML elements contain other XML elements, (i.e., sub-elements). We consider sub-elements and attributes as semantically the same although attributes have no hierarchical structures. In fact, elements are restricted by conditions specified on their sub-elements and attributes. XQL queries produce a set of elements satisfying conditions. Assuming that an element Ea satisfies conditions Ca and C'a over an element variable a, we define the semantics of XQL queries as follows:

Query: *select a from a where Ca*
Semantics: { Ea | Ca}
Query: *select a from a where Ca and C'a*
Semantics: { Ea | Ca } intersection { Ea | C'a }

Of course, if "or" is specified instead of "and" in the second query, then "intersection" is replaced by "union" in the semantics.

Elements can contain more than one sub-element of the same type, such as authors of books. In that case, consider the following query:

```
select $book.title
from bib URI "www.a.b.c/bib.xml",
      book $bib.book
where $book.author.lastname
      ="Ishikawa" and
      $book.author.lastname ="Kubota"
```

The query result has the following structure:

```
<XQL:result>
  <title> ...
  </title>
  <title> ... </title>
  ...
</XQL:result>
```

Its semantics is as follows:

{book | book.author.lastname ="Ishikawa"} intersection {book | book.author.lastname ="Kubota"}

Although the condition of the above query seems nonsense superficially, its semantics is valid (i.e., a nonempty set) and the query returns books co-authored by Ishikawa and Kubota.

(2) Data constructor

XQL allows any combination of retrieved elements to produce new element constructs. The following query produces new elements consisting of titles and authors of books published by Prentice-Hall:

```
select result <$book.title, $book.author >
from bib URI "www.a.b.c/bib.xml",
      book $bib.book
where $book.publisher.name
      = "Prentice-Hall"
```

Here "<>" in a select-clause enclosing elements delimited by "," creates new XML elements of a specified construct such as author and title tags. New elements have a name result as follows:

```
<XQL:result>
  <result>
    <title>A First Course in Database Systems
    </title>
    <author>
      <firstname>Jeff </firstname>
      <lastname>Ullman </lastname>
      <office> Gates Building</office>
    </author>
    <author> ... </author>
  </result>
  <result> ... </result>
  ...
</XQL:result>
```

As a book has more than one author, each result element consists of a series of authors and one title. Of course, authors of results have further structures such as lastname and office. Like this, the select-clause allows extraction and combination of sub-elements at any level.

(3) Join

XQL joins different elements by comparing their values in a where-clause. The following query joins books published after 1995 and articles by authors as a join key within the same XML data:

```
select result <$article, $book>
from bib URI "www.a.b.c/bib.xml",
      article $bib.article, book $bib.book
where $book.author.firstname
      = $article.author.firstname and
      $book.author.lastname
      = $article.author.lastname and
      $book.@year > "1995"
```

The query result has the following structure:

```
<XQL:result>
  <result>
    <article year="...">
      <author> ...</author>
      <title> ... </title>
      <publisher> ...</publisher >
    </article>
    <book year="...">
      <author> ...</author>
      <title> ... </title>
      <publisher> ...</publisher>
    </book>
  </result>
  <result> ... </result>
  ...
</XQL:result>
```

We define the semantics of a join query as follows (Ca,b is a join condition):

Query: *select <a, b> from a, b where Ca,b and Ca and Cb*
Semantics: {{ Ea | Ca} product { Eb | Cb} | Ca,b}

(4) Regular path expression

XQL allows regular path expressions for retrieving slightly heterogeneous elements. The following query retrieves last name of authors whose office is either a whole house or a room in a building:

```
select result <$author.lastname>
from bib URI "www.a.b.c/bib.xml",
        author $bib.book.author,
        office $author.(office | office.room)
where $office = "245"
```

Here "office" is bound to "$author.office" or "$author.office.room". "|" denotes choice between more than one path. Like this, XQL allows the user to query against semi-structured XML data.

The query result has the following structure:

```
<XQL:result>
  <result>
    <lastname> ... </lastname>
  </result>
  <result> ... </result>
  ...
</XQL:result>
```

The above query is alternatively expressed like this:

```
select result <$author.lastname>
from bib URI "www.a.b.c/bib.xml",
        author $bib.book.author
where $author.(office | office.room) = "245"
```

We don't use tag variables introduced by XML-QL [3]. Instead, we allow regular expressions as path expressions so that the user can simulate tag variables by using element variables declared as regular path expressions. The following query retrieves authors of any material such as book and article whose name is Ishikawa.

```
select result <$anyauthor>
from bib URI "www.a.b.c/bib.xml",
        anyauthor $bib.%.author
where $anyauthor.lastname ="Ishikawa"
```

"%" denotes "wild card". "$bib.%.author" matches both of "book.author" and "article.author". The query result has the following structure:

```
<XQL:result>
  <result>
    <author> ... </author>
  </result>
  <result> ... </result>
  ...
</XQL:result>
```

Consider a more complicated example. The following query retrieves heterogeneous elements such as book titles published in 1995 with either authors or editors whose name is Smith:

```
select result <$any.title, $AorE>
from bib URI "www.a.b.c/bib.xml",
        any $bib.%,
        AorE $any.(author | editor)
where $any.@year= "1995" and
        $AorE.lastname ="Smith"
```

Here "$any.(author | editor) " matches path expressions such as "book.author", "book.editor", "article.author", and "article.editor". The query result has the following structure:

```
<XQL:result>
  <result>
    <title> ... </title>
    <author> ... </author>
    ...
  </result>
  <result>
    <title> ... </title>
    <editor> ... </editor>
    ...
  </result>
  ...
</XQL:result>
```

Note that multiple occurrences of "%" specified by "any" in a from-clause are supposed to be bound to the same path at the same time in a query. We are afraid that full regular expressions can cause extra burdens on servers, so we temporarily restrict expressions to combinations of "%" and "|".

(5) Text match

The user can do approximate search over texts by using wild card "%" in strings. The following query obtains titles and authors of books authored by "Ishi" something:

```
select result <$book.title, $book.author>
from bib URI "www.a.b.c/bib.xml",
     book $bib.book
where $book.author.lastname ="Ishi%"
```

(6) Access to ordered elements

XQL preserves orders of XML data specified in a query implicitly. Further, XQL allows indexed access to ordered elements by specifying an index *[i]*. The following query obtains only the first author and title of each book published by Addison-Wesley:

```
select result <$book.title, $book.author[0]>
from bib URI "www.a.b.c/bib.xml",
     book $bib.book
where $book.publisher.name
 ="Addison-Wesley"
```

The query result has the following structure:

```
<XQL:result>
  <result>
    <title> ... </title>
    <author> ... </author>
  </result>
  <result> ...</result>
  ...
</XQL:result>
```

We provide a special index *last* or "-1" to have access to last elements of an ordered set. Note that $book.author [last] denotes last authors of books.

(7) Ordering elements

XQL can explicitly order elements by other elements. For example, an orderby-clause sorts book publisher names and titles in an alphabetical order by publisher name as follows:

```
select result <$book.publisher.name,
       $book.title >
from bib URI "www.a.b.c/bib.xml",
       book $bib.book
where $book.author.lastname ="Ishikawa"
orderby $book.publisher.name
```

Book titles of result elements are not automatically nested. That is, each element consists of exactly one publisher name and one title as follows:

```
<XQL:result>
  <result>
    <name> ... </name>
    <title> ... </title>
  </result>
  <result>
    <name> ... </name>
    <title> ... </title>
  </result>
  ...
</XQL:result>
```

(8) Grouping

XQL can group any elements by other elements. The following query groups book titles for each book publisher:

```
select result <$book.publisher.name,
       $book.title>
from bib URI "www.a.b.c/bib.xml",
       book $bib.book
where $book.author.lastname ="Ishikawa"
groupby $book.publisher.name
```

Here, a groupby-clause indicates that result elements are grouped by book publisher name. Although the groupby query seems similar to the previous orderby query, book titles of result elements are automatically nested in groupby, unlike orderby. That is, book titles with the same publisher name are grouped into a set of elements as follows:

```
<XQL:result>
  <result>
    <name> ... </name>
    <title> ... </title>
    <title> ... </title>
    ...
  </result>
  <result>
    <name> ... </name>
    <title> ... </title>
    <title> ... </title>
    ...
  <result>
  ...
</XQL:result>
```

This operation is called nesting, which is denoted as nest({<publisher name, title>}, publisher name). So each element consists of one publisher name and more than one title. We define the semantics of the groupby query as follows:

Query: *select <a1, a2> from a where Ca groupby a1*
Semantics: nest ({<a1, a2> | Ca}, a1)

We don't use nested query for grouping unlike XML-QL [3].

(9) Set operation

We provide set operations union, intersection, and difference operated on sets of elements. Note that set operators allow sets with heterogeneous structures. The last query in (4) can be expressed alternatively by using a set operator union as follows:

```
(select result <$book.title, $book.author>
from bib URI "www.a.b.c/bib.xml",
      book $bib.book
where $book.@year= "1995" and
      $book.author.lastname = "Smith")
union
(select result <$book.title, $book.editor>
from bib URI "www.a.b.c/bib.xml",
      book $bib.book
where $book.@year= "1995" and
      $book.editor.lastname ="Smith")
union        ...
```

Its semantics is literally as follows:

Semantics: { Ea | Ca } union { Ea' | Ca'} union ...

(10) Join of data from multiple data sources

The user can specify join of heterogeneous XML data from different data sources indicated by different URIs. The following query produces book author name and income by joining social security numbers of book authors and taxpayers at different data sources indicated by different URIs such as *b* and *t*.

```
select result <$author.lastname, $t.income>
from b URI "www.a.b.c/bib.xml",
      t URI "www.irs.gov/taxpayers.xml",
      author $b.book.author
where $author.ssn=$t.ssn
```

The semantics of join of multiple sources is the same as that of join within the same source.

The query result has the following structure:

```
<XQL:result>
  <result>
    <lastname> ... </lastname>
    <income> ... </income>
  </result>
  <result> ... </result>
  ...
</XQL:result>
```

In general, there are two approaches to resolving heterogeneity in schemas of different databases: schema translation based on ontologies and schema relaxation based on query facilities. XQL takes the latter approach, that is, XQL uses regular path expressions and element variables to enable the user to retrieve multiple databases with heterogeneous schemas by a single query at one time because the regular path expressions can match with more than one path and the element variables can be bound to more than one path. Further, we allow well-formed XML data containing a set of heterogeneous element as a query result. We think that a simple solution to schema translation between heterogeneous DTD based on XSL (i.e., XSL Transformations) if there are any DTD available.

(11) Multiple binding

The user can have universal access to multiple data sources by binding a single element variable to multiple URIs. The following example retrieves books authored by the same author from two different sources by only one query at the same time:

```
select result   <$book.title, $book.author>
from bib URI "www.a.b.c/bib.xml"
      "www.x.y.z/bib.xml", book $bib.book
where $book.author.lastname ="Ishikawa"
```

Its semantics is as follows (a is bound to a1 and a2):

Query: *select a from a1 a2, a where Ca*
Semantics: { Ea1 | Ca1 } union { Ea2 | Ca2}

(12) Embedding query

The user can embed an XQL query in XML data although XML parsers must be extended to recognize XQL. Queries delimited by a reserved tag name "XQL" are evaluated to produce sets of elements. The following XML data result to a set of book titles published after 1995:

```
<books>
  <XQL>
   select $book.title
   from bib URI "www.a.b.c/bib.xml",
          book $bib.book
   where $book.@year > "1995"
  </XQL>
</books>
```

That is, the result has the following structure:

```
<books>
  <XQL>
    <XQL:result>
      <title> ... </title>
      <title> ... </title>
      ...
    </XQL:result>
  </XQL>
</books>
```

(13) Function definition

The user defines a function by specifying an XQL query in its body. The role of functions is similar to that of relational views. The following finds declared income of employees by joining two sets of elements passed as parameters Taxpayers and Employees:

```
FUNCTION findDeclaredIncomes (Taxpayers,
Employees) as
  (select result <$Employees.name,
        $Taxpayers.income>
  from Taxpayers, Employees
  where $Employees.ssn=$Taxpayers.ssn)
```

See (14) in this section for invocation of functions. Note that function definition by more general programming languages such as Java is one of open issues.

(14) Function invocation

XQL allows invocation of functions in a query.

```
  select result <@id =
PersonID($author.firstname, $author.lastname),
                $author.firstname,
                $author.lastname,
                publicationtitle  $title>
  from bib URI "www.a.b.c/bib.xml",
    any $bib.%, author $any.author,
    title $any.title
```

Here, in a select-clause, the user inserts to a new attribute *id* values of functions such as "PersonID($author.firstname,$author.lastname)" and introduces "publicationtitle" as a new tag as follows:

```
<XQL:result>
  <result id="...">
    <firstname> ... </firstname>
    <lastname> ... </lastname>
    <publicationtitle> ... </publicationtitle>
  </result>
  <result id="..."> ... </result>
  ...
</XQL:result>
```

(15) Namespaces and other XML standards

Element tags and attributes have namespaces corresponding to URIs specified in from-clause as default namespaces. However, the user can explicitly specify namespaces by prefixing tag and attribute names as follows:

```
select result  <$book.title, $book.author>
from bib URI "www.a.b.c/bib.xml"
   "www.x.y.z/bib.xml", book $bib.book
where $book.author.lastname ="Ishikawa"
     and $book.JPN:price < "10000"
```

In this query, only books with JPN:price (i.e., prices in a namespace specified by JPN) are retrieved.

Note that how to relate other related XML standards such as RDF, XML Schema, XSL, Xpointer, Xlink, and DOM to XQL is another open issue.

We classify the above functionality into two groups. We call functions (1), (2), (3), (4), (5), and (6) XQL core (level 1). We call the other functions XQL extensions (level 2) because the former is more basic than the latter.

Supported features	FJ XQL	MS XQL	XML-QL
Selection	Yes	Yes	Yes
Construction	Yes	No	Yes
Join	Yes	No	Yes
Regular path expression	Partially Yes	Partially Yes	Yes
String regular expression	Partially Yes	No	Partially Yes
Indexed access	Yes	Yes	Yes
Order by	Yes	No	Yes
Group by	Yes	No	Partially Yes
Set operation	Yes	Yes	No
Multiple data source join	Partially Yes	No	Yes
Multiple binding	Yes	No	No
Embedding	Yes	No	Yes
Function	Yes	Yes	Yes
Conformity to XSL	No	Yes	No
Nonprocedural-ness	Yes	Yes	No

Table 1. Comparison of XML query languages.

4. Conclusion

4.1 Comparison with related work

We have proposed XQL as a query language for XML data of continuity with database standards in particular, in syntactic constructs and nonprocedural-ness. We would like to activate emergence of a query language which is both syntactically and semantically more understandable. We compare our XQL with other proposals in order to clarify our advantage. XML-QL [3] has much functionality in common with our XQL, such as selection, construction, join, regular path expressions, string regular expressions, indexed access, order by, multiple data source join, embedding, and function definition and use. Unlike our XQL, however, XML-QL lacks some functionality such as set operations, multiple binding, and user-friendly grouby. In particular, the major drawback with XML-QL is that it is not necessarily nonprocedural unlike our XQL; its multiple conditions are assumed to be sequentially evaluated. This makes query formation rather complex and decreases much efficiency in application development. Another XQL of Microsoft and et al.[8] has common functionality with our XQL such as selection, regular path expressions, indexed access, set operations, function definition and use, and nonprocedural-ness. Unlike our XQL, however, it lacks rather basic functionality such as construction, join, string regular expressions, orderby, groupby, multiple data source join, multiple binding, and embedding although it focuses more on filtering a single XML document by flexible pattern match conditions similar to XSL. This requires the user to write application logic in addition to query formation and also decreases much efficiency in application development. We summarize results of comprehensive comparison of three XML query languages FJ XQL (our XQL), MS XQL (Microsoft and et al.), and XML-QL in table 1, including future extensions.

4.2 Implementation

We conclude this paper by describing the implementation and some feedback from the experiences.

First, we describe storage schema for XML data. We have explored approaches to mapping DTD to databases (RDB such as Oracle or ODB such as Jasmine [5]) and to implement an XQL processing system [7]. If any DTD or schema information is available, we basically map elements to tables and tags to fields, respectively. We call this approach *DTD-dependent mapping* (See Figure 2), where the user must specify mapping rules individually. Otherwise, we take a DTD-independent mapping or *universal mapping* approach (See Figure 3), which divides XML data into nodes and edges of an ordered directed graph and stores them into separate tables for nodes and edges with neighboring data physically clustered. Note that the *order* fields are necessary for providing access to ordered elements by index numbers. Further, we provide separate tables for nonleaf and leaf nodes. Identifiers, such as ID and IDREF, realizing internal links between elements are declared as attributes and are stored as Value of Attribute_Node. So references through

XML data		
Tag_name1	Tag_name2	...
Value	*Value*	...
...	...	...

Figure 2. DTD-dependent mapping.

Edge

Identifier	Order	Label_identifier	Child_Node_identifier
Edge_Id_value	*Number*	*Label_Id_value*	*Node_Id_value*
...	...	...	...

Nonleaf_Node

Identifier	Path_identifier
Node_Id_value	*Path_Id_value*
...	...

Leaf_Node

Identifier	Order	Value
Node_Id_value	*Number*	*Value*
...	...	...

Attribute_Node

Identifier	Label_identifier	Value
Node_Id_value	*Label_Id_value*	*Value*
...	...	...

Figure 3. DTD-independent mapping.

identifiers are efficiently resolved by searching node identifiers matching with specified identifiers in the Attribute_Node table. Please note that XLink functionality such as simple and extended links has not been incorporated into the current version of our XQL because our current XML parser doesn't understand syntax and semantics of XLink. We have a future plan to extend our XQL to conform to such XML-related standards.

We cluster data in node and edge tables on a breadth-first tree search basis. We have found this way of clustering contributing very much to reducing I/O cost. We also have found that *label_id* and *path_id* fields, but not fields for labels and paths themselves, reduce storage space and search time. Further, we have known from our preliminary experiments that the DTD-dependent mapping approach is mostly two times more efficient than the universal mapping. However, we have focused on more of our implementation efforts on the universal mapping approach. The reasons are as follows:

(1) The approach can free the burden of defining idiosyncratic mappings from the users.

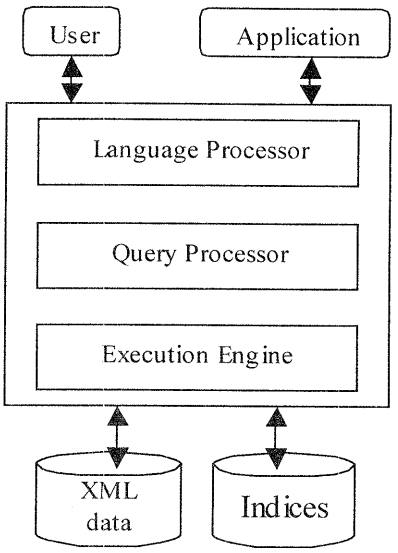

Figure 4. XQL Processing System Architecture.

(2) The approach can store XML data whose DTD are unknown in advance.
(3) The approach can store heterogeneous XML data, in particular, semi-structured XML data in the same database.

Next, we describe the system architecture for an XQL processing system (See Figure 4). We make appropriate indices on tag values, element-subelement relationships, and tag paths by pumping XML data from XML data sources in advance. We have understood from our empirical study that the multi-key indices such as (identifier, path identifier) for nonleaf nodes are better than alternative single-key indices such as (identifier) in our current system because of higher selectivity.

We describe how the XQL processing system works. The XQL language processor parses an XQL query and the XQL query processor generates and optimizes a sequence of access methods for efficient execution. In our current implementation of the universal mapping, the primitive access methods are basic operations on node sets, such as GetNodeIDbyPathIDandVAL, GetParentIDbyChildID, and GetVALbyNodeID, which are executed by the XQL execution engine. We have known that both RDB and ODB are usable as the underlying database systems of the XQL execution engine with the upper layers

unchanged only if the same set of the primitive access methods is dedicated to XQL execution engine by using the underlying database systems. In reality, we have implemented Oracle and Jasmine versions of XQL processing systems to make sure this fact.

Although we focus on the implementation of XQL processors on local XML data servers for the time being, we will approach to global XQL queries as follows: When the users issue a query against global servers for XML data, if the servers can understand XQL, the system obtains query results as XML data. Otherwise, it obtains whole XML data specified by URIs and processes the query against them locally. We think that detailed capability of XQL servers can be described by using RDF.

References

1) ANSI X3: SQL, *http://gatekeeper.dec.com/pub/standards/sql,* 1998.
2) Cattell, R.G.G. and Barry, D.K., Eds.: Object Database Standard: ODMG 2.0, Morgan Kaufmann Publishers, Inc., 1997.
3) Deutsch, A., *et al.*: XML-QL: A Query Language for XML, *http://www.w3.org/TR/1998/NOTE-xml-ql-19980819,* 1998.
4) Hagel III, J.H. and Singer, M.: Net Worth, Harvard Business School Press, 1999.
5) Ishikawa, H., *et al.*: An Object-Oriented Database System Jasmine: Implementation, Application, and Extension., *IEEE Trans. Knowledge and Data Engineering,* vol. 8, no. 2, pp.285-304,1996.
6) Ishikawa, H., *et al.*: *http://www.w3.org/TandS/QL/QL98/pp/flab.doc,* 1998.
7) Ishikawa, H., *et al.*: Document Warehousing Based on a Multimedia Database System, *Proc. IEEE 15th Intl. Conference on Data Engineering,* pp.168-173,1999.
8) Robie, J. *et al.*: XML Query Language (XQL), *http://www.w3.org/TandS/QL/QL98/pp/xql.html,* 1998.
9) W3C: W3C Query Language Workshop, *http://www.w3.org/TandS/QL/QL98/Overview.html,* 1998.

(Received September 20, 1999)
(Accepted December 27, 1999)

(Editor in Charge: *Kazumasa Yokota*)

Hiroshi Ishikawa received the B.S. and Ph.D degrees in Computer Science from the University of Tokyo in 1979 and 1992, respectively. He worked for Fujitsu Laboratories Ltd., from 1979 to 2000. He is now a professor of the Department of Electronics and Information Engineering at Tokyo Metropolitan University. His research interests include databases and e-commerce. He has published actively in international, refereed journals and conferences, such as ACM TODS, IEEE TKDE, VLDB, IEEE ICDE. He authored a book entitled *Object-Oriented Database System* (Springer-Verlag). He received the Sakai Memorial Distinguished Award from IPSJ in 1994 and the Director General Award from Science and Technology Agency in 1997. He is a member of IEEE, ACM, IPSJ, and IEICE.

Kazumi Kubota received the B.S. and M.S. in computer science from the Tokyo University of Agriculture and Technology. He joined Fujitsu Laboratories Ltd., in 1992. His research interests include databases. He is a member of IPSJ.

Yasuhiko Kanemasa received the B.S. in computer science from Tokyo Institute of Technology and M.S. in information science from Japan Advanced Institute of Science and Technology. He joined Fujitsu Laboratories Ltd., in 1998. His research interests include databases. He is a member of IPSJ.