An Active Web-based Distributed Database System for E-Commerce

Hiroshi Ishikawa **Manabu Ohta**

Tokyo Metropolitan University, Dept. of Electronics & Information Eng.

Abstract ECbusiness models like e-brokers on the Web use WWW-based distributed XML databases. To flexibly model such applications, we need a modeling language for EC businesses, specifically, its dynamic aspects or business processes. To this end, we have adopted a query language approach to modeling, extended by integrating active database functionality with it, and have designed an active query language for WWW-based XML databases, called *XBML*. In this paper, we explain and validate the functionality of XBML by specifying e-broker and auction business models and describe the implementation of the XBML server, focusing on the distributed query processing in the WWW context.

1 Introduction

XML data are widely used in Web information systems and EC applications. In particular, e-broker [12] business models on the Internet like Amazon.com, use a large number of XML data such as product, customer, and order data. In order both to flexibly model and agilely realize such applications, we need a modeling language for EC businesses, in particular, business processes. To this end, we will adopt a query language approach to modeling EC businesses, extended by integrating active database [8] functionality with it, and will provide an active query language for XML data centric in business models, tentatively called *XBML* (Xml-based Business Modeling Language) by extending the earlier version [10]. As an active query language approach, we need to consider its continuity with nonprocedural database standards such as SQL.

We rationalize the necessity of a modeling language for EC businesses. First, the modeling language must be able to integrate the components by reducing their complexity and to make the integrated system understandable. Second, the modeling language must be able to do more than model EC businesses. Indeed, we can use XML as interfaces of each component to make integration easy. However, this just models only the static aspects of the components. We must be able to model the dynamic aspects of business models, that is, business processes. For example, the author [4] discusses the necessity of modeling Web-based applications although he takes an HTML/JavaScript approach in the context of extending UML. But this approach would increase the complexity of modeling the business logic and the overhead of the client-server interaction on the contrary. Instead, we need a nonprocedural language approach to modeling the bushiness processes such as SQL as an analogical solution although just applying SQL to XML is inadequate because RDB and XML data have different data models. So we take a nonprocedural query language approach to modeling XML-based businesses. Further, we extend the query language approach by integrating ECA rules [8] with it for modeling control flow of the business processes. Thus, we call XBML an active query language approach to modeling EC businesses. At the same time, we make XBML efficiently executable on the server-side to agilely implement the business models.

This paper will not propose a new query language for XML submitted to W3C, although XBML contains the query functionality as a basic part for specifying business processes. XBML integrates the query facility with ECA rules for controlling business processes. We will describe the functionality of XBML and validate the usability of XBML by specifying the example business model with XBML in Section 2. We will describe the current implementation of an XBML server, focusing on the distributed query processing in Section 3.

2 Approach

2.1 Database Schemas and Business Model

We use the following database schemas or DTD fragments for illustrating the functionality of XBML:

```
<!ELEMENT dlib (book*, article*)>
<!ELEMENT book (author+, title, publisher, price, keyword*)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author+, title, publisher, keyword*)>
<!ATTLIST article year CDATA >
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname, office+)>
<!ELEMENT office (#PCDATA | (building, room))>
<!ELEMENT registration (register*)>
<!ELEMENT register (customer)>
<!ELEMENT customer (id, lastname, keyword*, purchased*)>
<!ELEMENT ordering (order*)>
<!ELEMENT order (id, item)>
<!ELEMENT shipping (ship*)>
<!ELEMENT ship (id, status)>
```

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

```
<dlib>
 <book year="1993">
   <author>
     <firstname>Hiroshi</firstname>
     <lastname>Ishikawa </lastname>
     <office>
       <building> L2 </building>
       <room> S210 </room>
     </office>
   </author>
   <title>Object-Oriented Database System
    </title>
   <publisher> Springer Verlag </publisher>
    <price> 69.00</price>
 </book>
</dlib>
```

We take an ordered directed graph as a logical model for an active query language XBML as a modeling language of EC businesses. That is, the data model of the XBML can be represented as data structures consisting of nodes (i.e., elements) and directed edges (i.e., contain, or parent-child relationships), which are ordered.

We also use e-broker business models based on XML data for describing the XBML functionality. Here we will provide the working definition to EC business models in general. The EC business models consist of business processes and revenue sources based on IT such as Web and XML. We assume that e-broker business models on behalf of customers consist of at least the following business processes:

(1) The customer searches products by issuing either precisely- or approximately-conditioned queries against one or more suppliers and /or navigating through the related links.
(2) The customer takes recommendations from suppliers into account if any.
(3) The customer compares and selects products and puts them into the shopping cart.
(4) The customer checks out by placing a purchase order with registration.
(5) The customer tracks the order to check the status for shipping.

The revenue source in e-broker models is sales.

2.2 Business Model Specification

We have adapted the design of XBML to the requirements for supporting EC business models discussed in the previous section.

(1) Searching Products

XBML provides the following functions for describing product search processes:

- XBML allows product search by *selecting* products based on their attributes, such as titles and authors, and *constructing* search results based on them.
- XBML allows ambiguous search by allowing *partially-specified string*s and *path expr*essions.
- XBML supports *data join* used in related search to promote cross-sell and up-sell.
- XBML configures search results by *sorting* and *grouping* them based on product attributes.
- XBML supports "comparison model" of similar products by allowing search *multiply bound* across shopping sites.
- XBML provides localized views (e.g., prices) of global products by introducing *namespaces* (i.e., contexts).

Note that we cannot describe sorting, grouping, and namespaces due to the limit of space.

Data selection and construction

The basic function of XBML is to select arbitrary elements from XML data by specifying search conditions for accommodating flexible product searches in e-broker business models. XBML allows any combination of retrieved elements to produce new element constructs for further services. The following query produces new elements consisting of titles and authors of books published by Prentice-Hall and firstly authored by Ullman:

```
(Query1)
  select result {$book.title, $book.author }
  from dlib URI "www.a.b.c/dlib.xml", book $dlib.book
  where $book.publisher.name = "Prentice-Hall" and
    $book.author[0].lastname ="Ullman"
    and $book.@year gt "1995"
```

The basic unit of XBML is a *path expression*, that is, an *element variable* followed by a series of tag names such as "$dlib.book". The user must declare at least one element variable in a from-clause. In particular, the user can bind XML data as input specified by URI to element variables such as dlib. Note that URI in our context must have the form *"www.x.y.z/d.xml"* but not *"www.x.y.z"*. This declares a context where an XBML query is evaluated. References of element variables are done

by prefixing "$" to them. The user checks a condition for selection in a where-clause. Two values of elements are compared in an alphabetical order. Compare operators include "=", "!=", "lt" for "<", "le" for "<=", "gt" for ">", and "ge" for ">=". XBML allows indexed access to ordered elements by specifying an index *[i]*. Attributes are referenced by prefixing "@" to them.

"{}" in a select-clause enclosing elements delimited by "," creates new XML elements of a specified construct such as author and title tags. The result of an XBML query is XML data, which can be retrieved as well as existing data. In our current design, the resultant XML data have no DTD, that is, they are well-formed XML data. For example, the result of the above query has the following structure, automatically wrapped by a tag "XBML:result":

```
<XBML: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>
  …
</XBML:result>
```

Here, we define the basic syntax of XBML as follows:

query = **select** target **from** context-list [where-clause] [orderby-clause] [groupby-clause]
target=expression | tag '{'expression-list '}'
expression-list = expression ','expression-list | expression
expression = [tag] '$' variable | [tag] '$' variable '.' path
path = '%' | tag | '@'attribute| path '.' path | '(' path '|' path ')' | text
context-list = context ',' context-list | context
context = variable* URI *uri-list | variable expression
uri-list = uri uri-list | uri
where-clause* = where *condition
condition = term | condition* or *term
term = factor | term* and *factor
factor = predicate |* not *predicate
predicate = expression compare expression
orderby-clause* = orderby *expression-list
groupby-clause* = groupby *expression-list

Partially-specified path expression

XBML allows regular path expressions for flexibly retrieving products represented as slightly heterogeneous elements (i.e., semi-structured XML data), which depend on data and suppliers in e-broker business models. Here we define semi-structured XML data as follows:

(1) Elements with the same tag are repeated at more than or equal to zero times, depending on parent elements, such as authors of books.
(2) Elements with the same tag have variant sub-structures, depending on parent elements, such as offices of authors.

As these characteristics cannot be determined in advance, we allow partially-specified path expressions.The following query retrieves authors of any material such as book and article named Ishikawa.

```
(Query2)
select result {$anyauthor}
from dlib URI "www.a.b.c/dlib.xml",
     anyauthor $dlib.%.author
where $anyauthor.lastname ="Ishikawa"
```

Here "%" denotes "wild card" in path expressions, which also allows approximate searches in e-broker business models. "$dlib.%.author" matches both of "book.author" and "article.author".

Data join

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

```
(Query3)
select result {$article, $book}
from dlib URI "www.a.b.c/dlib.xml", article $dlib.article,
       book $dlib.book
where $book.author.firstname    = $article.author.firstname
   and $book.author.lastname    = $article.author.lastname
   and $book.title = "%Electronic Commerce%"
```

In e-broker business models, this helps increase cross-sell and up-sell. Here the customers can do approximate searches over XML data by using wild card "%" in strings, that is, partially-specified strings, as is often the case with search in e-broker business models. The query result has the following structure:

```
<XBML:result>
  <result>
    <article year="…">
      <author> …</author>
```

```
        <title> … </title>
        <publisher> …</publisher >
      </article>
      <book year="…">
        <author> …</author>
        <title> … </title>
        <publisher> …</publisher>
        <price> … </price>
      </book>
    </result>
    <result> … </result>
    …
</XBML:result>
```

Multiple binding

The user can have universal access to multiple data sources by binding a single element variable to multiple URIs (i.e., URI list) in a where-clause. The following example retrieves books authored by the same author from two online bookstores (bound to *dlib*) by only a single query at the same time:

```
(Query4)
select result   {$book.title, $book.author}
from dlib URI "www.a.b.c/dlib.xml" "www.x.y.z/dlib.xml",
                book $dlib.book
where $book.author.lastname ="Ishikawa"
```

The users need to declare the partially-specified path expression to accommodate the heterogeneity of datasources. This function is necessary for,comparing similar products or searching the lowest price in multiple stores.

(2)Recommendation

Related search as a recommendation process is crucial in promoting cross-sell and up-sell, indeed. It is classified into three categories to the extent to which the customer in session is involved.

(1) *Non-personalized recommendation*

The customer is not involved. The e-broker recommends some products as general trends, independently of the customer. Or, the e-broker shows the customer products highly rated by the other customers.

(2) *Personalized recommendation*

The customer only is involved. The e-broker recommends some products based on the customer's psycho-graphic data, such as interests, or historical data, such as purchase records.

(3) *Collaboratively filtered recommendation* [17]

Both the customer and the others are involved. The e-broker recommends products purchased by those customers who purchased the products selected by the customer.

The facility for *function definition* and the *query transformation* technique have an important role in recommendation as follows.

Function definition

Functions correspond to "parameterized views". Functions modularize recurring queries in EC business models to increase their reuse. The user defines a function by specifying an XBML query in its body. The syntax has the following form:

***function-definition* = function *name* '(' *parameter-list* ')' as '(' *query* ')'**

parameter-list = parameter ',' parameter-list | parameter

As personalized recommendation, the following function recommends products based on the keywords which the customer (specified by its identifier, *customerid*) have registered in advance as his psycho-graphic data:

```
function personalized-Recommendation (customerid) as
(select result {$book.title, $book.price}
from dlib URI "www.a.b.c/dlib.xml" , book $dlib.book, r URI
"www.a.b.c/registration.xml", customer $r.register.customer
where $book.keyword = $customer.keyword and $customer.id
   = customerid)
```

The next example in the collaboratively-filtered recommendation category recommends products based on similarity that there are other customers who purchased the product selected by the customer (i.e., indicated by *selected*).

```
function collaboratively-filtered-Recommendation (selected) as
(select result {$book.title, $book.price}
from dlib URI "www.a.b.c/dlib.xml" , book $dlib.book, r URI
"www.a.b.c/registration.xml", customer $r.register.customer
where $book = $customer.purchased and
$customer.purchased = selected)
```

Query transformation

Until now, we have treated recommendation and search as separate processes. However, when the customer specifies search keywords, the search result can be expanded to include recommended products by transforming the original search query. Query transformation is classified into two rules as follows:

(1) Keyword addition rule

This rule has the general form:

```
keyword1 ==> keyword1 | keyword2
```

For example, the originally specified keyword "Electronic Commerce" adds a new keyword

"Internet Business" and the disjunctive condition is added to the end of the query as follows:

```
(Query5)
  select result {$book}
  from dlib URI "www.a.b.c/dlib.xml, book $dlib.book
  where $book.keyword = "Electronic Commerce" or
    $book.keyword = "Internet Business"
```

This technique is similar to query expansion [3] used in information retrieval. Note that this type of transformation keeps data sources unchanged.

(2) Data source addition rule

This rule uses set operations on queries to modify the original one. The rule has the following general form:

```
query1 ==> query1 set-operator query2
```

Here set-operator includes union, intersection, and difference. For example, when the customer searches books on EC, he will search articles on EC at the same time by modifying the original query with a disjunctive query as follows:

```
(Query6)
select result {$book}
from dlib URI "www.a.b.c/dlib.xml, book $dlib.book
where $book.keyword = "Electronic Commerce"
union
select result {$article}
from dlib URI "www.a.b.c/dlib.xml, article $dlib.article
where $article.keyword = "Electronic Commerce"
```

We analyze the application-based Web access patterns [5] to create the transformation rules, not discussed here due to space limitation.

(3)Moving to Carts

In general, EC business models involve temporary data, such as search results and shopping carts, valid only within sessions as well as permanent data such as books and customers. XBML handles such temporary data as first-class citizens.

Use of query results

XBML allows a query against the intermediate query results as well. The customer checks the result of searching products or recommendation to place an order. The following XBML query moves only the customer-checked items in the search result to the shopping cart:

```
(Query7)
select cart {item $result.book}
from XBML:result URI "www.a.b.c/XBML:result.xml" ,
    result $XBML:result.result
where $result.checked ="yes"
```

(4)Placing Orders

Selected items in the shopping cart remain to be added to ordering databases. Thus, addition of new elements is a mandatory function for constructing practical e-broker models. Addition of new elements often needs making them unique by invoking a dedicated function, defined in programming languages such as Java. To this end, XBML also allows *function invocation* in a query.

Insertion and function invocation

We provide the syntax for insertion by using an XBML query as follows:

insertion = ***insert into*** *target query*

The following query places a purchase order in e-broker business models by consulting the current shopping cart and customer data and invoking a function:

```
(Query8)
  insert into $order
  select order {@id = OrderID($customer.id, date()),
        item $cart.item}
  from r URI "www.a.b.c/registration.xml",
    customer $r.register.customer,
    XBML:result URI "www.a.b.c/XBML:result.xml" ,
    cart $XBML:result.cart, o URI "www.a.b.c/ordering.xml",
    order $o.order
  where $customer.lastname ="Kanemasa"
```

Here, in a select-clause, function calls "OrderID($customer.id, date())" generate unique order numbers. Ordering initiates internal processes, such as payment and shipment, hidden from the customers. Please note that "$order" in the into-clause is permanent in "www.a.b.c/ordering.xml" while "order" in the select-clause is temporarily constructed in this query.

(5)Tracking Orders

Ordering and shipping constitute a supply chain in the EC business models. Further, shipping is often outsourced. Thus, the involved data are managed at separate sites whether on intranet or on the extranet. To this end, XBML allows data *join across different sites* in addition to that within one site.

Join of data from multiple data sources

The user can join heterogeneous XML data from

different data sources indicated by different URIs. In e-broker business models, the following query produces a set of ordered items and shipping status by joining order identifiers of order entry data and order shipping data at different sites indicated by separate variables bound to multiple URIs, such as *o* and *s*:

```
(Query9)
select result {$order.item, $ship.status}
 from o URI "www.a.b.c/ordering.xml", s URI
    "www.d.e.f/shipping.xml", order $o.order, ship $s.ship
 where $order.id=$ship.id and $order.id="cidymd"
```

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. XBML takes the latter approach, that is, XBML 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. Of course, we admit that a simple solution to schema translation between heterogeneous DTD is based on XSL (i.e., XSL Transformations).

2.3 Applicability to Other Models and Extension

In the previous subsection, we have discussed the applicability of XBML to the e-broker models. Now we ascertain its applicability to business models other than the e-broker model. Indeed, there are rather novel EC business models, such as the reverse auction model. However, new business models are often created by mutation of business processes of existing models. We take the auction model [12] as an example. The auction model consists of the following processes:

The selling customer registers auction items.

(1) The buying customer searches auction items.

(2) The buying customer takes recommendations into account if any.

(3) The buying customer bids.

(4) The winner customer checks out by placing a purchase order with registration.

(5) The winner customer tracks the order to check the status for shipping.

We can observe similarity between the auction model and e-broker model. Registry of auction items by sellers corresponds to registry of products by suppliers, just implicit in the e-broker model. Searching and recommendation of auction items are very close to those of products in the e-broker model. Indeed, bidding is a new process, but it can be viewed as a series of tentative ordering until the buying customer wins the auction. In other words, the event that the customer wins the auction moves auction items to the shopping cart. The winner's placing a purchase order is very close to that in the e-broker model. Order tracking in the auction model is analogous to that in the e-broker model although it may require a new business model, such as e-escrow, to guarantee the bargain contract. The revenue source is a part of the contract price as fees in the auction model. Thus, we would say that our XBML can apply to the auction model as well.

However, it is also true that controlling business processes, or modeling events by some ways is necessary. Thus, the auction model requires triggering business processes at a specified time or on some database events such as insert. Active databases or ECA (Event-Condition-Action) rules [8] will be able to specify such business processes on events more elegantly than procedural programming languages plus the current version of XBML. Therefore, we extend current XBML by introducing the following construct for ECA rules:

on *event* **if** *condition* **then** *action*

Events include operations of XBML (e.g., select and insert) and a specified time. Conditions are specified as conditions of XBML. Actions are also specified by XBML.

For example, we think of the situation that when the highest bidding price of the auction specified by *id1* is updated, if the current time is before the closing time of the auction, then the auctioneer specified by *id2* increases his bidding by a specified value *value3*. The corresponding ECA rules can be specified as follows:

```
on insert into $auction.price
      if now() lt $auction.closing-time
      then insert into $auction.auctioneer.price
            select increase ($autioneer.price, "value3")
            from actn uri "www.a.b.c/actn",
                  auction $actn.auction
            where $auction.id = "id1"
              and $auction.auctioneer.id = "id2"
```

Here, *now()* returns the current time and *increase(var, val)* increments the variable *var* by a value *val*. The

ECA rules are defined in advance and invoked on events. The ECA rules can elegantly implement the recommendation (e.g., Query5):

```
on select result {$book}
        if $book.keyword = "Electronic Commerce"
        then select result {$book}
             from dlib URI "www.a.b.c/dlib.xml,
                book $dlib.book
             where $book.keyword =
                    "ElectronicCommerce" or
                $book.keyword = "Internet Business"
```

Note that the result of the "event query" (i.e., first "select") is replaced by that of the "action query" (i.e., second "select") in this case.

3 Implementation

XBML is intended for use in not only modeling EC business models, but also realizing them agilely. XBML must be efficiently implemented, too. XBML containing URIs intrinsically requires distributed query processing. So we construct the XBML server as follows:

(1) We construct local XBML servers as a basis.
(2) We construct global XBML servers by extending the local servers with server-side scripting techniques.

3.1 Local Server

We describe the basic architecture and implementation of a local XBML server. First, we describe storage schema for XML data. We have explored approaches to mapping DTD to databases (RDBMS, i.e., Oracle and ODBMS, i.e., Jasmine [9]) and to implement an XBML processing system [11]. 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*, where the user must specify mapping rules individually. Otherwise, we take a DTD-independent mapping or *universal mapping* approach, 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. We provide separate tables for nonleaf and leaf nodes. The *order* fields of Leaf_Node and Edge tables are necessary for providing access to ordered elements by index numbers. Identifiers, such as ID and IDREF, realizing internal links between elements are declared as attributes and are stored as Value of the separate Attribute_Node table. So references through identifiers are efficiently resolved by searching node identifiers in Attribute_Node.

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. Further, we have known from our preliminary experiments that the DTD-dependent mapping approach is mostly two times more efficient than the universal one. However, we have focused on more of our implementation efforts on the universal mapping approach for the following reasons:

(1) The approach can free the burden of defining idiosyncratic mappings from the users.
(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 a local XBML server or an XBML processing system. We make appropriate indices on tag values, element-subelement relationships, and tag paths in advance.

We describe how the XBML processing system works. The XBML language processor parses an XBML query and the XBML query processor generates and optimizes a sequence of access methods for efficient execution. The primitive access methods are basic operations on node sets, implemented by using RDBMS or ODBMS. They include get_NodeId_by_Path&Val, get_ParentId_by_Child, get_ChildId_by_Parent, get_Value_by_Id, get_NodeId_by_Path, and get_LabelId_by_LabelText in addition to node set operators, such as union, intersection, and difference. We illustrate the translation by using the query:

```
select $book.title
from dlib URI "www.a.b.c/dlib.xml", book $dlib.book
where $book.publisher.name = "Prentice-Hall"
```

This is parsed into an internal form, which denotes a logical query plan represented as an ordered-graph:

```
(Proj (Sel $book (Op_EQ $book.publisher.name
"Prentice-Hall")) $book.title)
```

Here, *Sel*, *Proj*, and *Join* (not in the above example) denote selection, projection, and join of XML data, respectively. *Op_EQ* denotes "=". This internal form is reorganized in a pattern-directed manner, such as placing *Sel* before *Join*, and is transformed into the following primitive operations:

(1) get_NodeId_by_Path&Val (Op_EQ "$book.publisher.name" "Prentice-Hall") returns a node set *set1 (i.e., $book.publisher)*.
(2) get_ParentId_by_Child (*set1* "$book") returns a node set *set2*.
(3) get_ChildId_by_Parent (*set2* "$book.title") returns a node set *set3*.
(4) get_Value_by_Id (*set3*) returns a value set as a result.

Both RDBMS and ODBMS can be used as the database system of the XBML processing system with the upper layers unchanged by virtue of the above primitive operators.

We describe the implementation of ECA rules. First, we define an event query by using the event and the condition in the rules and define an action query by using the action in the rules. Further, we store a dedicated ECA rule database whose entry consists of a pair of such an event and an action query. Now we consider the following approaches to ECA rule:

(1) Monitor-based approach

The monitor checks each usual query against the event query patterns in the ECA rule database and issues the corresponding action query of the matched event query if the condition is satisfied. The monitor usually keeps a queue of events generated by the matched event query and invokes the action query by looking up in the queue.

(2) Query rewriting approach

We modify the query processing. The parser checks each query against the event query patterns in the ECA rule database and recursively processes the corresponding action query of the matched event query by adding a check on the condition. That is, the "event query" and "action query" in the rules are translated into a sequence of queries (i.e., primitive access methods) with a condition check.

Next we consider the merits and demerits of the above approaches as follows:

(1) Monitor-based approach

The monitor can control the whole processes in a centralized manner. However, we need a monitor itself as an extra mechanism. It is not trivial to provide the facility for executing the event and action queries as a single transaction.

(2) Query rewriting approach

We need no extra mechanism for controlling processes.. It is rather straightforward to execute the event and action queries as a single transaction. However, the generated queries tend to be long, in particular, for cascading events.

For the moment, we adopt the query rewriting approach in favor of the ease of the implementation.

3.2 Global Server

Now we construct the global XBML server by extending the above local XBML servers with server-side scripting techniques. We provide preliminary definitions to queries. First, we categorize queries as follows:

(A) *Single-URI query*

This type of query contains only one XML data source specified by a single URL in the query, such as *Query1 (selection)* and *Query3 (join)*.

(B) *Multiple-URI query*

This type of query contains multiple XML data sources specified by multiple URIs in the query. This type is further categorized into two as follows:

(B1) *Decomposable query*

This type of query can be decomposed into a combination of single-URI queries with set operators, such as *Query4 (multiple binding)* and *Query6 (set operators)*.

(B2) *Non-decomposable query*

This type of query cannot be decomposed into a combination of single queries alone. This type of query contains join queries over multiple URIs, such as *Query9 (join of multiple data sources)*.

Second, we categorize queries in another way:

(a) *Local query*

XML data sources specified by URI are inside the relevant XBML server.

(b)*Global query*

XML data sources specified by URI are outside the relevant XBML server.

Now we show that non-decomposable (i.e., intrinsically global) query can be transformed into a series of single URL local or global queries and local queries (join). We assume that the original query contains n URIs. We translate a non-decomposable query by two steps:

(1) create a single-URI (local or global) query for each of n URIs with the insertion of the query result into the local server.
(2) create single-URI queries performing join of

the results stored in the local server, which are local queries, by reducing all URIs to a single-URI.

Queries generated by the step (1) localize single-URI global queries. Of course, single-URI local queries remain local. We call them localized single-URI queries. After that, queries generated by the step (2) simulate join of multiple data sources by join of local data sources. We call them localized join queries. For example, when we assume that the global server is resident at the "shipping site", consider again the following query (*Query9):*

```
select result {$order.item, $ship.status}
 from o URI "www.a.b.c/ordering.xml", s URI
    "www.d.e.f/shipping.xml", order $o.order, ship $s.ship
 where $order.id=$ship.id and $order.id="cidymd"
```

The query is translated into the following localized single-URI query, whose result is fetched into the global server:

```
select $order
 from o URI "www.a.b.c/ordering.xml", order $o.order
 where $order.id="cidymd"
```

and into the following localized join query, which produces a result of the original query:

```
select result {$order.item, $ship.status}
 from XBML:result URI "www.d.e.f/XBML:result.xml" ,
 order $XBML:result.order, s URI "www.d.e.f/shipping.xml",
 ship $s.ship
 where $order.id=$ship.id
```

Now we describe the global query processing, assuming that a query Q with a uri *URI* is specified as the input:

```
if Q is a single-URI query then
      process-or-dispatch (Q);
   else {/*i.e., Q is a multiple-URI query; */
      if Q is a decomposable query then
         { for each sub-query Qsub in Q
              process-or-dispatch (Qsub);
          merge the result by the local server;}
       else {/*i.e., Q is not a decomposable query;*/
          decompose Q into localized single-URI queries
              Qloc-s and localized join queries Qloc-j;
          for each sub-query Qsub in Qloc-s
              process-or-dispatch (Qsub);
          process Qloc-j by the local server;}
          }
       process-or-dispatch (Q) /* for single-URI query*/
        { if URI is local to the server then
            process Q by the local server;
          else {/*i.e., Q is not local to the server; */
            dispatch Q to the relevant remote server;
            store the result into the local server;}
```

}

The above query processing has some room for improvement in performance. Thus, if the non-decomposable query has no selection conditions, the whole remote data sources specified by the generated single-URI queries must be copied to the local server. For example, consider *Query9* when the global server is resident at "ordering site" or at a third site. Of course, if there is any selection condition on the join key, the condition is propagated to all the single-URI queries. We call this technique *simple selection condition propagation*. It is a kind of static query rewriting. However, we want more improvement. So we refine the process-or-dispatch scheme to sort the result of the query and return the value range with respect to the join key (i.e., MIN and MAX values) by adding "order-by" to the query.

Then, the conditions *"join-key ge min-value and join-key le max-value"* are dynamically added to the subsequent generated single-URI query. In turn, the query is evaluated to produce a new value range of the join-key (i.e., min-value' and max-value'). The following characteristic holds: *min-value' >= min-value & max-value' <= max-value*. From this, we can conclude that the expected selectivity is better than that of the original algorithm. If a single-URI query Q has any selection condition "key_Q *ge* $min\text{-}value_Q$" and "key_Q *le* $max\text{-}value_Q$", then we take $MAX_Q(min\text{-}value_Q)$ and $MIN_Q(max\text{-}value_Q)$ as an initial min-value and max-value, respectively. A single-URL query being firstly processed is chosen from ones with any selection condition on the non-key because now all the single-URL queries virtually have the same condition on the key (i.e., the initial value range). If there is no selection condition, any local query being firstly processed will produce the initial value range. It has the merits: It can avoid extra data transfer by just issuing modified queries and avoid extra protocol by just accommodating the min/max values in results.

XBML works as server-side scripting with database access such as CFML[2], and ASP [15] and provides universal access to distributed XML data. If XBML queries are embedded in XML-based scripts, the global XBML server can provide more direct and universal interfaces to representing and accessing distributed XML data than the other approaches. That is, XML pages containing the element <XBML> *XBML-query* </XBML> are interpreted as scripts.

4 Conclusion

We have proposed and validated XBML as an XML active query language approach to specifying EC business models. We compare our work with related work. There are no high-level language approaches to modeling EC business processes, in particular, no other work on validating the modeling language by applying it to EC business models. XBML can provide a more direct and universal tool for modeling distributed XML data applications than server-side scripting tools such as CFML [2], ASP [15].

Now we will compare our XBML with other query language proposals from the viewpoint of process specification since XBML contains the query language functionality as a basic part. XML-QL [6] has comprehensive functionality and has much in common with our XBML. However, condition specification in XML-QL is rather verbose. If applied to business modeling, XML-QL would make query formation rather complex.

XQL [16] has compactly-specified functionality and has common functionality with our XBML. XQL focuses more on filtering a single XML document by flexible pattern match conditions similar to XSL. If applied to specifying EC business models involving multiple sites, XQL would require the user to write extra application logic in addition to query formation.

Lore [7] provides a powerful query language for retrieving and updating semi-structured data based on its specific data model OEM, but it lacks some functionality such as multiple binding.

So far we have compared XBML with the other works only from the viewpoint of query languages. However, the above languages are largely different from XBML for the following reasons. First, we focus our efforts on the distributed query processing in the Web context. However, the above works don't cover such a topic. Second, we think that the functionality of ECA rules is mandatory in order to model control flow of E-businesses. However, all of the above query languages lack ECA rules.

Web query languages, such as W3QL [13], view the Web as a single huge database and enable to address the structures and contents. XBML views a single Web source as a database and allows queries over Web-based distributed databases.

The active views [1] focuse on the comprehensive functionality of ECA rules. On the other hand, we have concluded the necessity of ECA rules from the experiences of applying XBML to concrete businesses. We extend the query optimization in relational databases[14] to the distributed context.

References

1 Abiteboul, S. et al.: Active Views for Electronic Commerce, Proc. Intl. Conf. VLDB 1999

2 Allaire Corporation: CFML, http://www.allaire.com/documents/cf4/CFML_Language_Reference/contents.htm, 2000

3 Chang, C.H., et al.: Enabling Concept-Based Relevance Feedback for Information Retrieval on the WWW, IEEE Trans. Knowledge and Data Eng., vol.11, no. 4, pp.595-609, 1999

4 Conallen, J.: Modeling Web Application Architectures with UML, Comm. ACM, vol.42, no.10, pp.63-70, 1999

5 Cooley, R., et al.: Web Mining: Information and Pattern Discovery on the World Wide Web, Proc. the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97), 1997.

6 Deutsch, A., et al.: XML-QL: A Query Language for XML, http://www.w3.org/TR/1998/NOTE-xml-ql-19980819, 1998

7 Goldman, R., McHugh, J., and Widom, J.: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language, Proc. the 2nd Intl. Workshop on the Web and Databases (WebDB '99), 1999.

8 Ishikawa, H., et al.: An Active Object-Oriented Database: A Multi-Paradigm Approach to Constraint Management, Proc. Intl. Conf. VLDB, 1993

9 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

10 Ishikawa, H., et al.: http://www.w3.org/TandS/QL/QL98/pp/flab.doc, 1998

11 Ishikawa, H., et al.: Document Warehousing Based on a Multimedia Database System, Proc. IEEE 15th Intl. Conference on Data Engineering, pp.168-173, 1999

12 Jutla, D., et al.: Making Business Sense of Electronic Commerce, IEEE Computer, pp.67-75, Mar. 1999

13 Konopnicki, D. et al.: W3QS: A Query System for the World-Wide Web. Proc. Intl. Conf. VLDB, 1995

14 Makinouchi, A. et al.: The Optimization Strategy for Query Evaluation in RDB/V1, Proc. Intl. Conf. VLDB, 1981

15 Microsoft: ASP, http://www.activeserverpages.com, 2000

16 Robie, J., et al.: XML Query Language (XQL), http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998

17 Special Section: Recommender Systems, Comm. ACM, vol.40, no.3, pp.56-89, 1997