Customized Content Delivery through XML Message Brokering

R. Gururaj, and P. Sreenivasa Kumar

Indian Institute of Technology Madras, Chennai, India

Summary
XML is being accepted as a standard format for representation and exchange of web data. XML message brokers play a key role as message exchange points for messages sent between producers and consumers. An XML message broker can perform filtering, transformation, and routing of received messages. In certain applications, XML message brokers may need to perform advanced customization (value-addition) where modifications to the structure and content of the original message are done as per the preferences of individual clients. Customizing the content of messages is desirable and significant in the context of personalized content delivery, data and application integration, and co-operation among disparate web services. At present, XML message brokers support user profile matching and limited customization only. In this paper, we propose a system named **VAXBro**, a value-adding XML message broker that addresses the data processing needs of value-addition process in an XML message broker. In this work, we also discuss the proposed customization service specification language, XML update approach and process optimization techniques.

Key words:
Message customization, XML, message broker, application integration.

Introduction

Information Dissemination involves distributing data produced by data sources to a set of interested data consumers in a distributed environment. A *message broker* (MB) plays a key role as central exchange point for messages sent between message sources and consumers. In this article we deal with *MB*s that adopt *publish-subscribe* (PS) information dissemination model. The PS model is a specialization of information dissemination protocol with push-based, aperiodic data-delivery mechanism. In this paradigm, submitted user preferences are called *profiles*. The pub-sub system accepts *profiles* from the end-user, and collects new information/data from different sources and matches received information against profiles and updates the user with relevant information. As *Extensible Markup Language* (XML) has emerged as a standard for representation and exchange of data on the web, we assume that the future dissemination systems support XML data exchange. A message broker that handles XML data is called an *XML message broker*. The important functions of an XML message broker are: (a) *filtering* incoming messages against large number of user queries to find if the message matches the user requirements, (b) *transformation* that restructures the message according to the user requirements, and (c) *routing* which involves transmitting the message to the user. The above mentioned transformation activity can customize the message under distribution, to facilitate data and application integration, and personalized content delivery. This kind of customization is different from web-personalization, which refers to the action that changes the layout and content of the web page according to the user preferences. But the personalization in XML message brokers deals with customization of data contained in XML format, which is meant for processing; not just viewing as in the case of HTML.

In the recent past, many XML filtering systems have been proposed [1][2][8][9]. All these systems accept user interests (longstanding profiles) in the form of XPath [14] expressions and perform filtering on arrival of a message, to find the set of user queries that match the incoming message. A recent XML message broker proposed in [3] accepts user queries in XQuery [14] format and provides slightly advanced customization functionality where the result of a user query is delivered to the user with customized tags.

In customized content delivery scenario, it is appropriate to have XML message brokers that support more advanced transformations that satisfy the needs of end user. Next, in the context of application integration, it is natural to see applications that are developed independently in a distributed environment. In such cases, applications may be dealing with XML data that have structural disparity. When these applications are integrated, one application may send data to the other application for the purpose of further processing. Because of the structural mismatch, the data received can't be used directly. If the data is transformed to suit the needs of receiving application, then application integration becomes more straightforward and effective.

Manuscript received May 5, 2006.
Manuscript revised May 25 , 2006.

We call the customization action that supports sophisticated transformations to XML data as *value-addition*. This paper attempts to address the needs of *value-adding XML message brokers* that support *value-addition* to XML messages under dissemination.

The rest of the paper is organized as follows. In Section 2, we discuss related work and motivation. Section 3 provides details about value-adding service specification and proposed XML update mechanism, and Section 4 describes the system design and architectural features of VAXBro. Implementation details are given in Section 5. In Section 6, we present performance results with process optimization techniques and, finally, Section 7 concludes the paper.

2. Related Work and Motivation

Value-addition to an XML message may include actions such as- 1) inserting new elements into original message, 2) performing some complex/simple computations on data items to produce new data items for inclusion in original message or delivering to user with specified tags, 3) separate a portion of the document to produce new XML fragments, 4) delete some elements from original message, and 5) aggregation (merging) of information received from different sources.

Now, let us have a look at the functionalities of current XML message filtering/broker systems. XFilter [1] provides an efficient matching of XML documents to a large number of user profiles, which are in the form of XPath expressions. The XFilter enforces XML filtering by converting XPath queries into a set of finite state machines (FSM), which react to the XML parsing events. For each XPath we have an FSM. The XFilter matches the incoming document against all user profiles. On matching, entire document is sent to the user. The functionality of another XML filter named WebFilter [9] is similar to that of XFilter except the matching process. The user profiles are XPath [14] expressions and stored in the system as attribute value pairs. The XML messages received from the sources are matched against all the user profiles.

YFilter [2] is an improvement over its earlier version-XFilter. YFilter represents all the user queries as a single non deterministic automata (NFA), as against separate FSM for each path in XFilter. This shared processing of XPaths improves the performance.

An XML message broker described in [3], extends the YFilter matching functionality and provides slightly advanced functionality. The user interests are in the form of XQuery queries. The result of the XQuery query is wrapped in an XML fragment with user specified tags. Though it is possible to specify delete and rename using XQuery return clause, it is not natural and straightforward. And, the work described in [3], doesn't support computation, and merging of data extracted from another XML document into the original message.

Another most recent XML filtering system named FiST [8], performs XML filtering by sequencing twig patterns. In this system, XPaths with node tests are evaluated faster than that of in YFilter.

We observe that none of the present XML message brokers support value-added customization. The need for advanced customization (value-addition) and inadequacies of the present XML brokering systems stimulated us to propose a system named VAXBro: a value-adding XML message broker that addresses the data processing needs of value-addition process in XML message brokers. The major issues in realizing VAXBro are- (i) customization service specification, (ii) storing and updating XML messages, and (iii) process optimization. In the following sections, we discuss all the above-mentioned issues in detail.

As our proposed system transforms the structure and content of the original message, the data processing needs are entirely different from earlier systems.

3. Value-addition Service Specification and XML Updates

In this section we present the proposed value-addition service specification language and our proposed XML update approach, which is effective in the context of XML message brokering.

3.1 Value-addition Semantics

A document in a message broker could be received from an external source, or available locally in the broker. User queries specify customization actions on one of the incoming documents and may involve one or more local or other incoming documents. Each customization activity can be thought of as a *value-adding service* (VS). User specified *VS* may involve a set of *operations*. The following are the abstract customization actions: (i) *update* (INSERT, DELETE and RENAME), (ii) *compute*, and (iii) *return*. Our proposed *value-addition* assumes the following combinations of above mentioned actions: (a) {*update*}: modify and send the entire document, (b) {*update-return*}: modify and return a portion of the document, (c) {*compute-update-return*}: compute a value, modify and return a portion of the document, (d)

```
< stockQuotes >
   < stock >
        < symbol >  </symbol >
        < price>    < /price>
        <time>      </time>
        < date>     < /date>
   < /stock>
   . . .
< /stockQuotes>
```

(a) stockQuotes.xml

```
< companyProfiles>
   <company>
        < symbol >  </symbol >
        <name>     </name>
        < ceo >    </ceo>
        < hq>      < /hq >
        < address > < /address >
   < /company>
    . . . .
< /companyProfiles >
```

(b) companyProfiles.xml

```
< orgonization>
   < orgo >
        < name >  </name >
        < url>       < /url>
   < /orgo>
   . . .
< /orgonization>
```

(c) orgo.xml

Fig. 1 Sample XML documents.

{*compute-return*}: compute and return the result, (e) {*compute-update*}: compute a value and modify the document for dissemination, and (f) {*return*}: return a portion of the document. Keeping the above value-addition actions and system characteristics in mind, we have proposed a value-addition service specification language.

3.1 Customization Service Specification Language

We propose a customization specification language, which extends subset of XQuery [14] and its update extensions [13], as shown in Figure 2 (a) and (b) respectively. Our language supports variable binding, variable assignment, arithmetic functions, and return constructs of XQuery, along with INSERT, RENAME and DELETE semantics of XQuery update extensions as discussed in [13].

Each user query may contain computation, update and return statements. All operations specified in update clause, and the return statement, are considered as *suboperations* (SOP) of the query. We have discussed the preliminary issues related to the proposed customization service specification language, in our work [4] and [5]. Here, we give full details.

3.2 Grammar for Service Specification Language

The proposed service specification language supports the important XQuery and update features like: (1) FLWR expressions, (2) Update clause with insert, rename and delete operations, and (3) simple aggregation and arithmetic operations. The grammar for the proposed service specification language is shown in Figure 3. Our proposed language is not a replacement for earlier XML query formats, and it is not a general purpose XML query language. It is proposed to suit XML message brokering model.

```
For $b IN
document("stockQuotes.xml")//stock
WHERE $b/symbol='IBM'
RETURN  <stockDetails>
                {$b}
          </stockDetails>
```

(a)

```
FOR $binding1 IN XPath-expr,…
LET $binding := XPath-expr,…
WHERE predicate1,….
UpdateOp,….

EBNF for UpdateOp:
UPDATE $binding { subOp {, subOp}*}

and subOp is:
DELETE $child | RENAME $child TO name |
INSERT content [BEFORE | AFTER $child] |
REPLACE $child WITH $content |
FOR $binding  IN XPath-expr,…
        WHERE predicate1,.. UpdateOP
```

(b)

Fig. 2 XQuery and its update syntax.

In the proposed language, all XPath variable bindings are done in one FOR clause and a query can have more than one LET clause. More than one UPDATE clause is possible. All node tests are done in FOR clause itself. The proposed specification doesn't handle ATTRIBUTES, IDREFs, nested FOR, nested RETURN statements. The INSERT operation performs append only. Advanced features like - ORDER BY, GROUP BY etc., are not supported. A sample service specification (user query) is shown in Figure 4.

```
serviceRequest::=((forClause)|(letClause)) (letClause)* (updateOp)*   (returnExpr)?

forClause::=FOR{varBind {,varBind}*}
varBind::=$binding IN XpathExpr
letClause::=LET{$binding:=asst {, $binding:=asst}*}
asst::=(mathExpr | mathFunction | otherAsst)
mathExpr::=(XpathExpr | numericConst) (op) (XpathExpr | numericConst)
op::=(+|-|*|/)
mathFunction::=(avg|sum|min|max) "((" ({(XpathExpr | numericConst)
                        {,(XpathExpr| numericConst)}*}) "))"
otherAsst::= (XpathExpr | numericConst)

updateOp::= UPDATE $binding "{" {subOp {, subOp}*}  "}"
subOp::= (DELETE $child) | (INSERT ((tcontent)|(XpathExpr)) |
                                                (RENAME $child TO name)
tcontent::= (tag) ((contentString) | (XpathExpr)) (endTag)
returnExpr::=RETURN (taggedContent)
taggedContent::=(tag)(taggedContent | ( "{" (XpathExpr) "}" ) )+ (endTag)
tag::= "<" tagName ">"
endTag::= "</" tagName ">"
```

Fig. 3 Grammar for the proposed language.

The query in Figure 4 customizes the incoming XML message- *stockQuotes.xml*, shown in Figure 1 (a), which contains stock information of some listed companies. This query inserts *ceo* information for the given company, extracted from *companyProfiles.xml* (Figure 1 (b)), into *stockQuotes.xml*. Then it renames the element *price* to *cost* and deletes the *date* element.

```
FOR $st IN document("stockQuotes.xml")/stockQuotes,
      $sc IN $st/stock,
      $pr IN document("companyProfiles.xml")/
           companyProfiles/company[symbol=$sc/symbol]

UPDATE $sc
   {
   INSERT $pr/ceo
   RENAME $sc/price TO 'cost'
   DELETE $sc/date
   }
```

Fig. 4 Customization request.

3.3 XML Update Approach

In *VAXBro*, all XML messages are stored in BDB XML [10] database. BDB XML is an embedded database to manage and query XML documents. BDB XML stores documents in native form where the logical structure of the document is retained. BDB XML can be used through programming API. As BDB XML is an embedded engine, and can be used with application in the same way as we would use any other third-party package. In BDB XML documents are stored in container, which we create and manage through *XmlManager* objects. Each such object can open multiple containers at a time. Each container can hold a large number of XML documents. Once the document is placed into a container, we can use XQuery to retrieve documents or required parts of documents. Queries are evaluated through *XmlManager* objects. BDB XML supports XQuery working draft. As XQuery is an extension to XPath2.0, BDB XML provides full support for that query language also.

As the present XQuery draft doesn't include update features, BDB XML too doesn't support modifications to XML through XQuery. But, BDB XML provides document modification facility through its API. This allows us to easily add, delete, or modify selected portions of XML document.

Our proposed XML update technique was presented in our earlier work [6]. The following discussion explains our approach to updating XML data in VAXBro, using BDB XML API.

A. INSERT operation: this is to add an element to the document at specified location. We use the method *addAppendStep()*. The parameters need to be passed are: (i) location where the said content is to be inserted, (ii)

content to be inserted, and (iii) the tag name of the inserted content.

B. RENAME operation: this operation changes the name of a tag of an element specified by the path. For this we use the method - *addRenameStep().* This method requires the element path to be renamed and the new name.

C. DELETE operation: will remove all elements addressed by the path. This is executed by *addRemoveStep()* method. This method requires the element to be deleted.

All the above methods are defined in *XmlModify* class of BDB XML API. It is clear that all the required parameters to call the methods *addAppendStep(), addRenameStep()* and *addRemoveStep(),* are directly or indirectly available in the specified query.

Once, we add modification steps by calling appropriate methods, finally we call the method *runModify()* of *XmlModify* object by passing on the document to be modified and other required parameters. In our approach, first we parse the user submitted customization query and extract all the required information to call BDB XML API methods, and store the same in suitable data structures. As all the queries in message brokers are longstanding queries, this parsing activity is done only once for each query. Next, whenever it is needed to customize the documents in BDB XML database, our query engine (Operation Handler) calls appropriate methods of BDB XML API, by passing on required information, stored in local data structures. The entire update handling process is depicted in Figure 5.

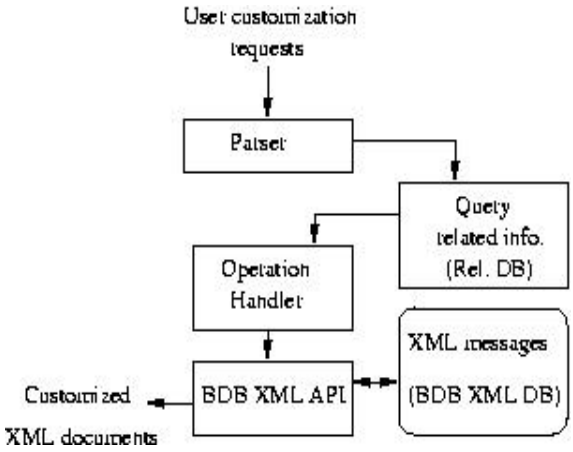


Fig. 5 Handling XML updates.

For clarity, let us consider the following query, which involves XML documents with structures shown in Figure 1.

Query:

```
FOR $st IN document('stockQuotes.xml')/
                                    stockQuotes/stock,
$pr IN document('companyProfiles.xml')/
        companyProfiles/ company[symbol=$st/symbol]
UPDATE $st
{
        INSERT $pr/ceo
        RENAME $st/symbol TO 'org'
        DELETE $st/price
}
```

The above query has one INSERT operation where the element *ceo* from *companyProfile.xml* is inserted into *stockQuotes.xml*. Here, the target location where insertion takes place is *$st*, which is bound to *document('stockQuotes.xml')/stockQuotes/stock* path, the content to be inserted is specified by *$pr/ceo* and is bound to */companyProfiles/company/ceo* path in *companyProfiles.xml*, and the name of the tag is *ceo* by default, because it is not explicitly mentioned by the user.

Next, for RENAME operation we use the method- *addRenameStep().* This method requires the element path to be renamed and the new name. In the above query, the second SOP is a RENAME operation. The element to be renamed is specified by the path *$st/symbol*, and the new name is *org*. Similarly, the DELETE operation is executed by the- *addRemoveStep()* method. This method requires the element to be deleted. In our example query, it is given by the path *$st/price*.

3.4 Update Performance Comparison

Here, we compare the performance of our proposed update approach with another approach where Relational Database Management System is used to store XML data. And this study is made with an intention to analyze the effectiveness of *relational approach* and *our approach* in the context of XML message brokering. The relational approach is discussed in [11][12][13].

Basic features of relational approach are implemented and a set of four queries was considered for test execution, which perform INSERT, DELETE and RETURN operations. INSERT operation inserts a new arbitrary element with user specified content. DELETE removes a specified leaf element. RETURN operation returns a specified leaf element. The above set of queries was executed on both the implementations. The customization time for the tested queries, for both the approaches is shown in a graph as given in Figure 6. From the results it is evident that our approach to updating and rebuilding XML documents works better than that of relational

approach. In the present relational implementation we have not implemented rename operation, inserting complex elements, and inserting elements extracted from other documents. It is obvious that the complexity of such operations would further reduce the performance in case of relational approach. This is because, new tables are to created for new elements, which are not in old structure. Further, in message brokering systems, after modifications, the document need to be built back for dissemination.

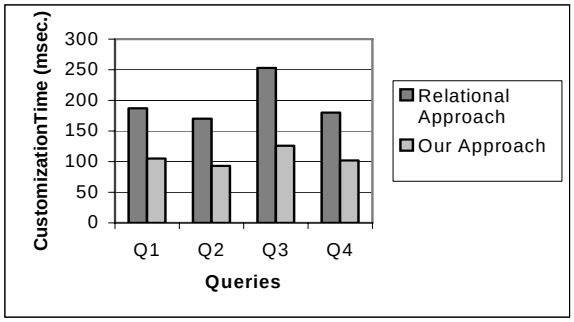


Fig. 6 Comparison of Update performance.

In relational approach, many tables are to be accessed and hence, cumbersome. In our approach, this is straightforward because documents are stored in native form. Another disadvantage with relational approach is that, every time a new message arrives, we need to shred that data into respective tables before doing any processing. This is not needed in proposed approach. And, if multiple queries request different customization, it is too complex to handle that kind of situation in relational approach, as we need copies of same set of tables for each query. In proposed approach, we just copy the whole target document for each query. Hence, our proposed approach to storing and modifying XML data is more effective than relational approach for XML message brokering systems.

4. System Design and Architecture

Our proposed value-adding XML message broker *VAXBro*, accepts user customization requests in a format as discussed in the earlier section. On receiving a message from a source, the system executes all the involved user queries. We call the incoming document on which customizations are specified as *target document*. System output consists of a set of customized XML documents, which will be delivered to concerned users. The consumer of the customized message could be an end-user (human or an application that accepts XML data input from external sources), or another XML message broker on the downstream of information dissemination network.

The major activities of *VAXBro* are: (a) query management, (b) message management, (c) operation handling, and (d) dissemination management. The core of *VAXBro* functionality is as follows: given a large set of customization requests specified using the proposed service specification language, and a collection of local documents, perform modifications and restructuring of incoming XML messages using effective data processing schemes. The query management and message handling are independent of each other. One specific requirement of the system is that each query needs customizations on a target document in its own way. Hence, multiple copies of the same target document are needed.

We have presented the architecture of *VAXBro* in our earlier work [8]. The proposed architecture is shown in Figure 7. The important components and their functionality is as discussed below.

A. Data store

This consists of two databases. The first one is Berkeley DB (BDB) XML [10] database to store XML documents, and the second is a relational database to store query, message and user related information.

B. Query manager

This module collects all the queries submitted by users and parses them to extract customization (suboperations) and user-query related information. The parsed information is stored in relational database in appropriate form. This module is also responsible for extracting and storing query containment information, which is needed for process optimization to be discussed in later sections.

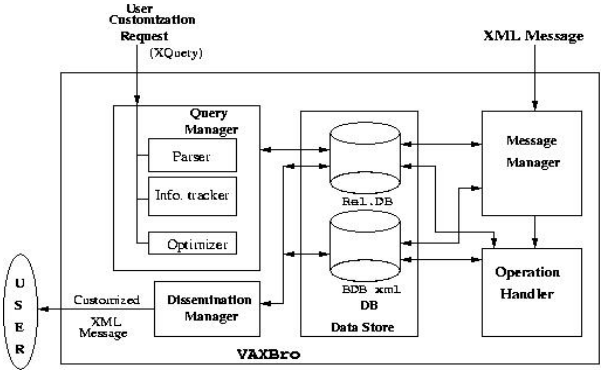


Fig. 7 VAXBro architecture.

C. Message manager

Is responsible for receiving a message from a source and storing the same in XML database. Based on the stored

query related information, this module finds the queries to be triggered for the incoming message. The execution of involved queries is triggered only if all involved documents are available at the system. This module invokes the functioning of operation handler module. As required, multiple copies of the target document are made and stored in XML database.

D. Operation handler

This module executes the SOPs of a query in specified order, to produce customized content (results) for dissemination. This module extracts the details (stored in relational database) required to execute individual SOPs. The queries are executed on specified copy of the target document designated for the query. Finally, the customized output of the value-adding process is stored in XML database with a specified name.

E. Dissemination manager

This takes care of disseminating the customized messages to concerned users. Once, the customized output is made available by the operation handler, the dissemination manager module extracts the concerned user information for each query executed by the operation handler (this information is available in relational database) and delivers the output document to the same. Once the output is sent to the concerned users/clients, the result document is deleted from the XML database appropriately.

We propose to store the customization information contained in a query, per suboperation basis, in relational database. The operation handler module executes SOPs of each query in specified order, with the help of this information. Our system executes SOPs of a query individually. This process gives rise to incremental transformation of the message.

5. Implementation Details

The prototype implementation of the proposed system is complete. Our present implementation is based on the following assumptions: (a) customization operations are always targeted on incoming messages, not on local documents, (b) data updates on local documents are dealt separately, and for the sake of brevity, it is not discussed in this paper, (c) reducing processing time is more significant than reducing the space required by the process, (d) input XQuery queries are syntactically correct, (e) structure of all documents is known apriori, and (f) no document will have two versions at given time instance. Now, we discuss the implementation issues involved in each of the modules of *VAXBro*.

A. Data store

We have chosen to store query, message and user related information in IBM DB2 relational database. Storing query related information such as- suboperation details, target documents, and document copies on which the query is executed is more crucial. We have designed appropriate relational table structures to capture all the required data. To store XML documents we use BDB XML database. The advantages of using BDB XML are: (a) it supports XPath and XQuery expression evaluation, (b) its API provides methods to modify the structure and content of the document, and (c) we don't have to rebuild the document back, after modifications as it is done in the case of storing XML data in relations of some RDBMS. The BDB XML database supports storing the whole document as a single object.

B. Query manager

The query manager module is a Java program and calls a JavaCC (Java Compiler Compiler) [7] program to parse the input query. The following are some important things that are result of the parsing process: (1) user related information, (b) query related data, and (c) customization actions defined in a query (SOPs). For each SOP, it extracts the information like- target document, other documents involved, operation type, source content (in case of INSERT), computations involved, rename or delete information, and return statement fields. The information extracted depends on the action performed by the SOP. The extracted information is stored in appropriate table structures of the relational database. This module, also finds the query overlaps and records the same in appropriate format in relational database. The operation handler uses the overlap information at later point of time, to optimizing the process as discussed in previous section.

C. Message manager

Message manager is a Java program. On receiving a message, we extract the list of queries that can be triggered, and for each query we check if all the other involved documents (incoming or local) are readily available with the system. If ready, the message manager triggers the execution of that query by passing the query *id* to operation handler module. Otherwise, we just save the message in XML store and postpone the query execution. For each target document, the message manager creates copies with specified file names, per query basis.

D. Operation handler

Once, the query execution is triggered by the message manager, message processing is invoked by operation handler program written in Java. For the query under execution, it extracts all the suboperations of that query, from relational tables and then executes them in specified order. We plan to execute SOPs with the help of facilities

provided by BDB XML database version 2.0 for Java. BDB XML database supports XPath and XQuery. If the SOP is a FILTER, or COMPUTE with RETURN, we can straightaway execute the SOP, in XQuery format. If the SOP is a modification request, then we perform updates as discussed in Section 3. All required parameters to execute the modification methods are directly or indirectly available in relational tables of system data store. In our proposed prototype, we implement only few features of the proposed service specification language. And it is not difficult to understand that, more features can be implemented, as every piece of information, which is required to modify the document, or to query the document is available in some form in the input query. The *parser* of the query manager module does the extraction of all the required information from input queries.

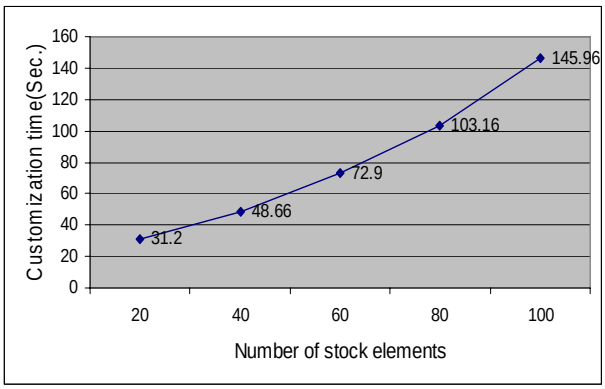
All suboperations of a query will work on same copy of the target document. Each query has one final output document. If intermediate result of a query is used by other queries, then intermediate output of the query maintained in the system after executing specified suboperations. As the focus of our work is on data processing needs of the system, we don't implement the functionality of the dissemination manager module. We assume that output XML messages are disseminated to the concerned users based on the user-query information stored in relational database. All output XML documents can be deleted once they are disseminated.

6. Performance

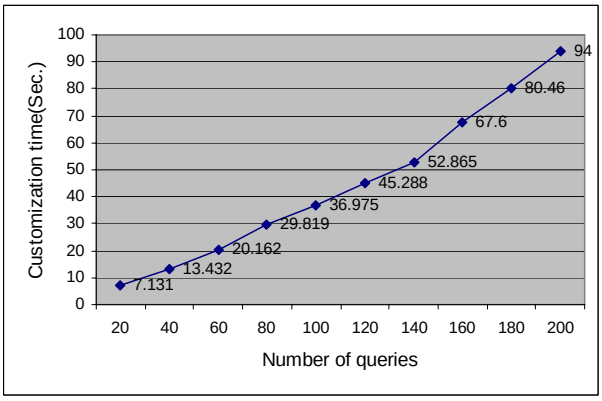
Having completed the implementation of the prototype, which is needed to validate our ideas, we conducted the following experiments to observe the performance of the system with basic data processing algorithm. Experiments were conducted on an Intel Pentium 4 machine with 1.7 GHz. and 256MB RAM, running on MS Windows2000.

The test data is stock market information being provided by YAHOO site. Stock details are refreshed once in every 20 minutes. For this set of experiments we use the following three XML documents- (1) *stockQuotes.xml* (contains stock price information), (2) *companyProfiles.xml* (contains company related information), and (3) *orgo.xml* (contains some other organizational data). We have written a program to pull stock price details from the site, and generate *stockQuotes.xml*. We consider both *stockQuotes.xml* as incoming documents, and *comapanyProfiles.xml*and *orgo.xml* as local document. The structures of the above mentioned documents are shown in Figure 1.

In Our first experiment, we have taken a set of 100 randomly generated queries, and run them against documents with varying size (varying the number of stock elements in each involved file). The time taken to execute the set of 100 queries for varying number of elements is shown in Figure 8(a). We observe that as the number of elements increase the customization time also increases. The rate of change is slightly more for larger number of elements.



(a) Customization time for varying number stock elements, & 100 queries.



(b) Customization time for varying number of queries, & 50 stock elements.

Fig. 8 System performance.

Second experiment was done by keeping the size of the involved documents constant and varying the number of randomly generated queries. The number of stock elements in documents was 50. The observations are plotted in graph as shown in Figure 8(b). We observed a moderate rise in customization time as the number of queries in the system is increased.

6.1 Process Optimization

Our proposed value-addition increases the processing overhead of the system considerably. We propose the following techniques to optimize the data processing in *VAXBro*.

A. Shared processing of queries by exploiting commonality

The proposed intermediate representation of the XQuery query splits the complex value-adding service into suboperations. This will give rise to incremental message transformation. For each user submitted query q_j, we conduct the containment test to find if: (a) q_j is equivalent to other user query i_j, in such case we just add the user *id* to the list of user *id*s for q_i in data structure where query and user mapping is stored, (b) the first *n* suboperations in q_j, are equal/similar to first *n* suboperations in q_i ; in such case we save a copy of the target document modified by the first *n* suboperations of q_i and that will become the target document for q_j. While executing q_j, we skip first *n* suboperations in. In such case we should always execute q_i before q_j and later one is dependent on the former one, and (c) the query is disjoint and for no *n*, first *n* suboperation of the query are similar to that of first *n* SOPs in any other query. In such case it has a separate entry and not dependent on any other query. Thus, we exploit the commonality among customization requests. The whole exercise is to avoid redundant operations and make use of intermediate results of one query by another. The above scheme is depicted in Figure 9. In the example shown in Figure 9, we have three queries- Q1 with SOPs-{a,b,c,d,e}, Q2 with {a,b,k,m}, and Q3 with {a,b,k,j,t}. SOPs with same alphabet represent identical operations and further we assume that all the queries modify same target document.

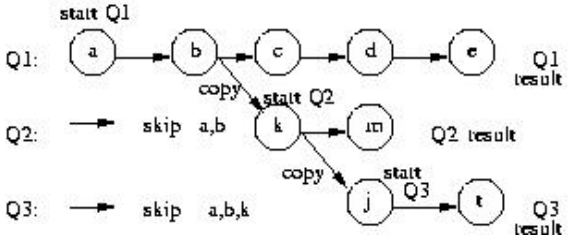
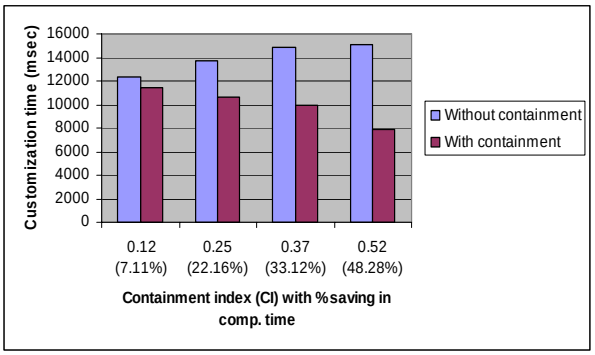
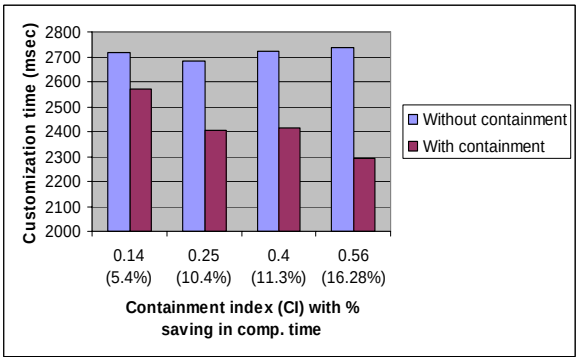


Fig. 9 Exploiting commonality among the user queries.

The above optimization technique is implemented in our prototype. Experiments were conducted to assess the effectiveness. During the experiment, different sets of queries were considered with varying degree of containment. We introduce the notion of *containment index* (CI)} to measure the degree of containment in a given set of user queries. This CI varies between 0 and 1. Lower CI represents less containment. We conducted experiments with two target documents one with 20, and the other with 100 stock elements in *stockQuotes.xml*. The graphs shown in Figure 10, show the results of experiments on containment. On x-axis, we have CI along with percentage of savings in computation time on exploiting the containment. We observed that performance of the system improves with increase in containment and increase in target file size.



(a) With 20 stock elements



(b) With 100 stock elements

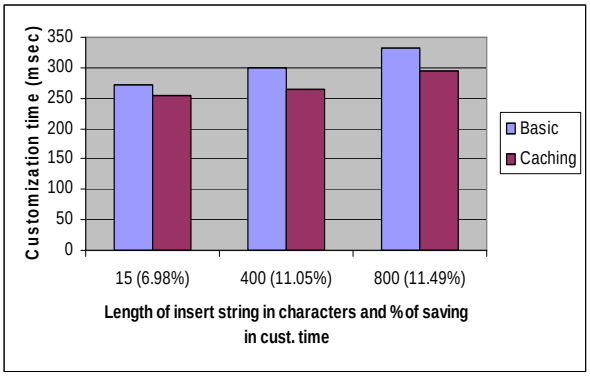
Fig. 10 Performance improvement on exploiting commonality among the user queries.

A. Reusing pre-computed results of XPaths

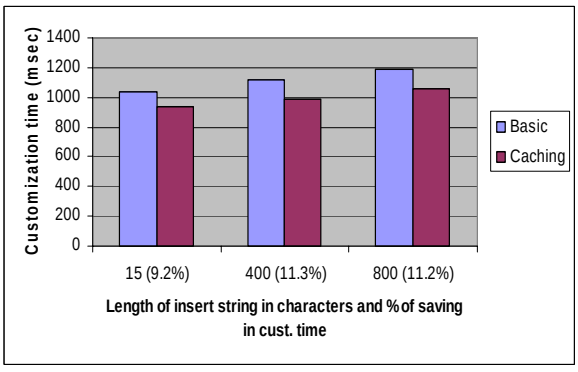
Our second technique is based on exploiting the static nature of the local documents in our system. We know that the data contained in local documents are relatively static. It doesn't change as frequently as incoming data. Queries may involve SOPs that extract data from local documents and insert the same into some target document. In such case we plan to evaluate the XPath expressions that involve local documents only once when they are

encountered for the first time and store the result in appropriate data structures. During the subsequent encounters, we don't reevaluate them. Instead, we just use the data, which has been cached earlier. We illustrate the above with an example query shown below, which involves an SOP that inserts *ceo* information of companies extracted from *companyProfiles.xml* into *stockQuotes.xml*.

```
FOR $st IN document("stockQuotes.xml")/stockQuotes,
$sc IN $st/stock,
$pr IN
document(“companyProfiles.xml")/companyProfiles/comp
any[symbol=$sc/symbol]
UPDATE $sc
 {
   INSERT $pr/ceo
 }
```



(a) With 20 elements in *stockQuotes.xml*

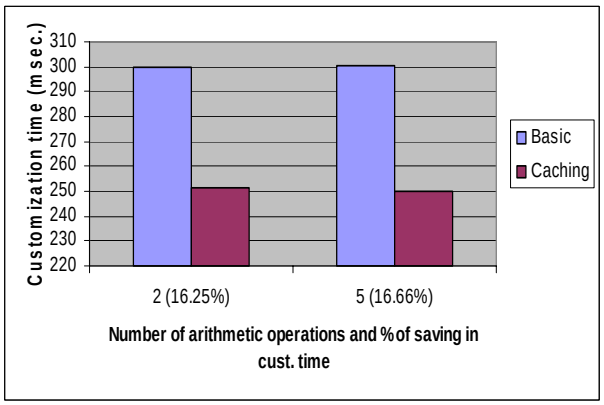


(b) With 100 elements in *stockQuotes.xml*

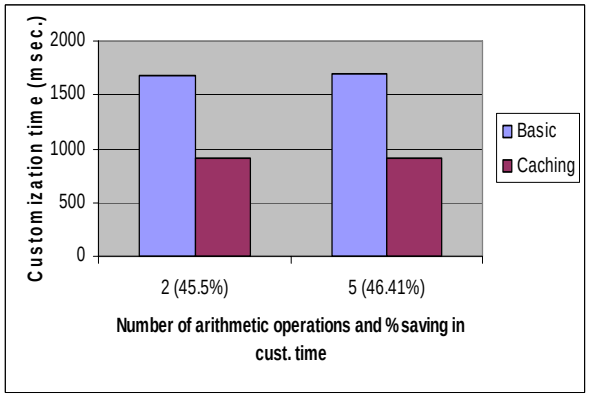
Fig. 11 Performance improvement on using pre-evaluated results of XPath expressions on local documents.

Here, we assume that the document *companyProfile.xml* is local to the system and contains data, which is relatively static. We evaluate the XPath *$pr/ceo* on *companyProfile.xml* with appropriate bindings during the first evaluation and store the outcome along with insert key (used to join elements from two documents) information in appropriate data structures. During the subsequent requirements we use the stored data for insertion operations.

The above technique is also implemented in prototype. We experimented with insert string of varying lengths (i.e., 15, 400, and 800 characters), and varying target document size (20 and 100 stock elements in *stockQuotes.xml*). Results of our experiments are shown in Figure 11. We observe that the saving in customization time is more if the length of the insert string (result of XPath exression on local document) is more. Figure 12 shows the improvement in performance on using pre-computed results of arithmetic operations involving data from local documents. The saving in customization time is more for larger no. of operations and larger document size.



(a) With 20 elements in *stockQuotes.xml*



(b) With 100 elements in *stockQuotes.xml*

Fig. 12 Performance improvement on using pre-evaluated results of arithmetic operations.

7. Conclusions

This paper presents the design and development of *VAXBro*, a value-adding XML message broker. *VAXBro* supports advanced customization to XML messages under dissemination. The focus of the paper is on addressing issues in customization service specification, XML updates and other data processing needs of *VAXBro*. The proposed approach to storing and updating XML documents using Berkeley DB XML native database works more effectively than *relational approach*, in the context of XML message brokering. The prototype implementation of the proposed system is complete. Further, experiments were conducted on prototype, in order to test the correct functioning of the system and to assess the behavior and performance of the system at various workloads. The following process optimization techniques are proposed and tested: (1) exploiting the commonality among the user queries (shared processing), and (2) use of pre-evaluated XPath results on static documents. Both the optimization techniques showed considerable improvement in performance. Observations made during the experiments are reported with appropriate representations. Because of its inherent complexity and sophisticated customization facilities, the proposed XML message broker- *VAXBro* is best suited for application integration than large-scale dissemination. There is a scope for extending the customization functionality of the system with more number of value-addition operation types like- replacing parts of an XML document with new content etc.

References

[1] Altnel M. and Franklin M.J. (2000). Efficient Filtering of XML Documents for Selective Dissemination of Information. *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 53-64.

[2] Diao Y., Franklin M.J. (2003). High-Performance XML Filtering: An Overview of YFilter. I*EEE Data Engineering Bulletin*, Vol. 26:1, March, 41-48.

[3] Diao Y., Franklin M.J. (2003). Query Processing for High-Performance XML Message Brokering. *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 261-272.

[4] Gururaj R. and Sreenivasa Kumar P. (2005). Service Specification in a Value Adding Broker for Data Dissemination Through XML. *Proceedings of the IACIS Pacific 2005 Conference*, Taipei, Taiwan, pp. 406-413.

[5] Gururaj R. and Sreenivasa Kumar P. (2005). VAXBro: A Value-Adding XML Message Broker. To appear in *Proceedings of the ADCOM 2005 Conference*, Coimbatore, India.

[6] Gururaj R. Giridhar Reddy M., and Sreenivasa Kumar P. (2006). An Effective Approach for Modifying XML Documents in the Context of XML Message Brokering. To appear in Proceedings of the IACIS Fall 2006 Conference, Reno, Nevada, USA.

[7] Java Compiler Compiler (JavaCC). Available: http://javacc.dev.java.net.

[8] Kwon, J., Praveen Rao, Bongki Moon, and Sukho Lee (2005). FiST : Scalable XML Document Filtering by Sequencing Twig Patterns. *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, pp. 217-228.

[9] Pereira J., Fabret F., Jacobsen H.A., Llirbat F. and Shasha D. (2001). WebFilter: A High-throughput XML-based Publish and Subscribe System. *Proceedings of the 27th VLDB Conference*, Rome, Italy, 723-724.

[10] Sleepycat software's Berkeley DB XML database. Available: http://www.sleepycat.com/products/xml.shtml.

[11] Shanmugasundaram, J., Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. (1999). Relational database for querying XML documents: Limitations and opportunities. *Proceedings of the 25th VLDB Conference*, Edinburgh, Scotland, 302-304.

[12] Shanmugasundaram, J., Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. (2000). Efficiently publishing relational data as XML documents. *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 65-76.

[13] Tatarinov, I., Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. (2001). Updating XML. *Proceedings of the ACM SIGMOD Conference*, Santa Barbara, CA, USA, 413-424.

[14] World Wide Web Consortium. Available: http:// www.w3c.org.

Author Profile

R. Gururaj : He is a Ph.D., scholar of Computer Science and Engineering Dept., at Indian Institute of Technology Madras, India. He received his Master of Technology (M.Tech.) from Birla Institute of Technology, Ranchi, India. His research areas include

databases, data integration and dissemination systems, and message brokers.

P. Sreenivasa Kumar : He is a professor in Computer Science and Engineering Dept., at Indian Institute of Technology Madras, India. He received his Master of Technology (M.Tech.) and Ph.D., in engineering, from Indian Institute of Sciences, Bangalore, India. His research areas include databases, data integration and dissemination systems, and semistructured data, and ontology.