An Efficient Co-operative Framework for Multi-Query Processing over Compressed XML Data

Juzhen He[1], Wilfred Ng[2], Xiaoling Wang[1], and Aoying Zhou[1]

[1] Department of Computer Science and Engineering, Fudan University, Shanghai 200433, China
{juzhenhe, wxling, ayzhou}@fudan.edu.cn
[2] Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong
wilfred@cs.ust.hk

Abstract. XML is a de-facto standard for exchanging and presenting information on the Web. However, XML data is also recognized as verbose since it heavily inflates the size of the data due to the repeated tags and structures. The data verbosity problem gives rise to many challenges of conventional distributed database technologies. In this paper, we study the XML dissemination problem over the Internet, where the speed of information delivery can be rather slow in a server-client architecture which consists of a large number of geographically spanned users who access a large amount of correlated XML information. The problem becomes more severe when the users access closely related XML fragments, and in this case the usage of bandwidth is inefficient. In order to save bandwidth and process the queries efficiently, we propose an architecture that incorporates XML compression techniques and exploits the results of XPath containment. Within our framework, we demonstrate that the loading of the server is reduced, the network bandwidth can be more efficiently used and, consequently, all clients as a whole can benefit due to savings of various costs.

1 Introduction

XML has become the standard for data exchange on the Web, and database servers are employed to store large amounts of XML documents, such as XML digital libraries and XML dissemination systems. In these applications, XML repositories are employed to support queries from many clients. However XML data are verbose due to their repeated tags and structures. Most XML documents in the servers are compressed in order to reduce the storage size. Many previous works have studied techniques for efficient evaluation of XML path expression [11, 14], and other works have focused on different kinds of XML compression technologies [1, 6, 9, 10, 12, 13, 19]. However, these works do not consider how to process compressed XML documents in XML subscribe/dissemination applications, such as the RSS (Really Simple Syndication) news distribution system,

which supports processing multiple queries imposed by a group of clients. Assume that there are co-operative relationships among clients, as Fig. 1 shows. The server keeps the compressed large-scale XML document, and clients cooperate to obtain information or news from the server. In this scenario, it is important to adopt distributed techniques and XML compression approaches to save bandwidth in result delivery. For example, the server is in London and users from Beijing pose queries to the server. After query processing and result publication, some results on the users' local machines may be reusable in response to the subsequent queries posed from Shanghai.

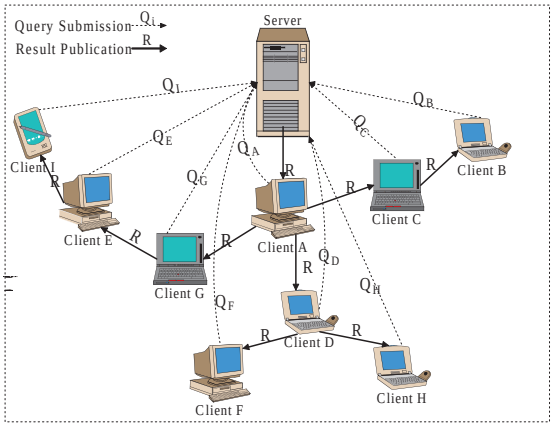

Fig. 1. Architecture of a Co-operative Framework

One might also find that distributed SQL query techniques in traditional RDB have been extensively studied [2] and applied in server-client architectures. However, these conventional database techniques are not directly applicable to distributed XML query processing, especially over compressed XML documents. To our knowledge, this is the first paper to address the problem of efficiently processing XML queries over a co-operative framework with XML compression techniques. The main contributions of this paper can be summarized as follows:

- We propose a co-operative framework for multi-query processing over compressed XML data. We study how to process these XML queries in Internet-scale XML data dissemination applications, such as the RSS news dissemination system.
- We exploit XML compression technology to reduce the system's bandwidth consumption. Though some previous works have studied various XML compression techniques, none of them has studied how to handle co-operative clients to gain efficient dissemination on the Web.
- We develop a special index structure QIT, which helps the server to process queries efficiently and helps clients to obtain results from compressed XML

fragments reserved by other clients. This technique is shown to benefit all clients as a whole, since the average network cost is greatly reduced.
- We carry out an empirical study. Our experimental results show that the proposed methods are efficient and practical.The loading of the server is reduced, the network bandwidth can be more efficiently used and consequently, all clients as a whole can benefit due to savings in various costs.

The rest of the paper is organized as follows. Related work is introduced in Section 2. Section 3 describes the preliminaries. Sections 4 and 5 present our approaches of building the index structure on the server side and processing queries over compressed XML. Section 6 gives the experimental results related to the efficiency of our framework. Finally, Section 7 concludes and discusses future work.

2 Related Works

Recently, several methods have been proposed for query optimization of an XML document [14], in which the structural index is an efficient approach for path/structure queries. There are also some research results [15, 16] that combine a structure index with keyword search for XML document retrieval. For example, the index introduced in [16] integrates both the advantages of a structural index and inverted lists. However, there is no previous work on efficient multi-query processing over compressed XML documents. We need to establish a better framework that is able to handle both structure and text queries in a smaller storage space. There are a few XML compression techniques [1, 6, 9, 10, 12, 13, 19] that can be classified into two categories according to whether the encoded document can be queried directly or not. XMill [6], which is an example of the first category, aims to minimize the size of the XML document as much as possible and achieves the highest compression ratio of all compressors. XGRIND [1] and XPRESS [9], two examples of homomorphism compressors in the second category, both support directly querying of compressed data by retaining the document structure after compression. XGRIND uses dictionary encoding and Huffman encoding for tags and data separately, whereas XPRESS adopts reverse arithmetic encoding and diverse encoding methods depending on the data type, which allows XPRESS to achieve a better compression ratio and higher query efficiency than XGRIND. In this paper, we design our framework based on XPRESS techniques to disseminate compressed XML documents over the co-operative server-client architecture.

3 Preliminaries

XPath, widely accepted as the core component of XML query languages, is adopted as our query language. We constrain our XPath in $XP^{\{/,//,*\}}$ in this paper. The grammar of $XP^{\{/,//,*\}}$ is given as:

$$q \rightarrow l|*|.|q/q|q//q \tag{1}$$

where "l" is the label of XML document, "$*$" is a wildcard and "." denotes current tag. "/" and "//" means child and descendant, respectively.

There exists containment relationships among different queries in $XP^{\{/,//,*\}}$, and it is necessary to exploit this containment relationship to speed up the publication of the query result. If query Q_A contains query Q_B after computing, we can send Q_A's result to corresponding client C_A and ask C_A to send Q_B's result to client C_B to avoid the server sending Q_B's result to both C_A and C_B. This approach reduces the server's load and saves the server's bandwidth.

Definition 1 (Containment of XPath). *For XPath Q_1 and Q_2, if the result of Q_1 is always contained in the result of Q_2 for every XML instance, we say Q_1 is contained by Q_2, and denote this fact as $Q_1 \subset Q_2$.*

The containment of the $XP^{\{/,//,*\}}$ expression is a CO-NP problem, and [7] gives an efficient but not complete PTIME algorithm to compute the containment. Each XPath expression can be expressed as a *one-arity* pattern tree, and vice versa (as Fig. 2 shows, p and p' are pattern trees of XP_p and $XP_{p'}$ respectively). Thus XPath expressions can be translated into tree patterns, and the containment is evaluated based on the homomorphism between the corresponding pattern trees. When there is a homomorphism between pattern trees, there also exists a containment relationship between these XPath queries.[7]

Definition 2 (Pattern Homomorphism). *For two tree patterns p and p', if there exists a homomorphism $h : p' \rightarrow p$, then $p \subset p'$.*

And one can determine in $O(|p||p'|)$ whether a homomorphism exists [7]. Fig. 2 is an example of one homomorphism from pattern tree p' to p. Thus, the query $XP_{p'}$ contains the query XP_p.

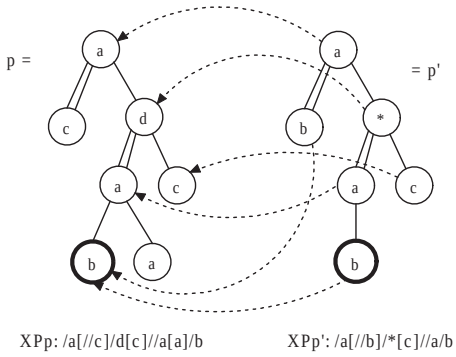

Fig. 2. A homomorphism from p' to p

Based on the containment relationship in Definition 2, we design a query-index in order to store these relationships. Fig. 3 shows that a query index tree is built on the server side to store this containment among queries.

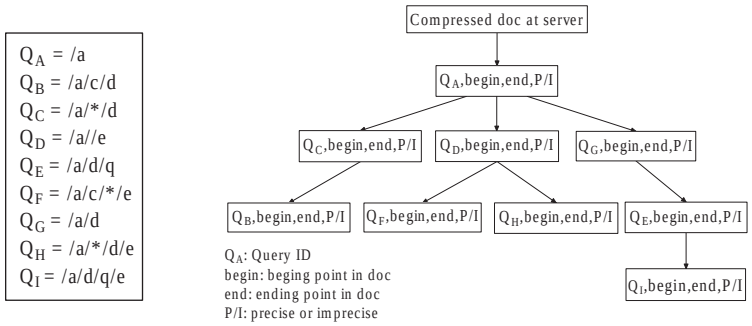

Fig. 3. An Example of QIT (Query Index Tree)

On the other hand, in order to minimize the document's size and save bandwidth, we adopt an XML compression approach. We choose XPRESS [9] to be the compression tool in our framework. We also extend "intervals" technique to speed up the query processing. The "intervals", which are used to encode tags in XPRESS, are helpful to process the query on the compressed document. The containment among "intervals" indicates the containment of the suffix for simple paths, thus "intervals" technique can be used in complex query processing. The interval of a tag is computed based on the probability of this tag in the XML document, and each tag in the document has a simple path that contains its parent and ancestors. For a simple path $/p_1/p_2/\ldots/p_n$, assuming that the probability of p_i is $prob_i$, the original interval before compression of p_i is $[MIN_{io}, MAX_{io})$ and the compressed interval is $[MIN_i, MAX_i)$, where

$$MIN_{io} = \sum_{k=1}^{i-1} prob_k, MAX_{io} = \sum_{k=1}^{i} prob_k \tag{2}$$

$$MIN_i = MIN_{io} + prob_i * MIN_{i-1}, MAX_i = MIN_{io} + prob_i * MAX_{i-1} \tag{3}$$

For example, there are "a", "b", and "c" three different elements in an XML document. Assuming their probabilities are 0.3, 0.3 and 0.4, and their original intervals before compression are $[0.0, 0.3)$, $[0.3, 0.6)$ and $[0.6, 1.0)$ respectively. For query Q_A: $//c$, whose interval is $[0.6, 1.0)$, and query Q_B: $/a/c$, with interval $[0.6 + 0.4 * 0.0, 0.6 + 0.4 * 0.3)$, that is $[0.6, 0.72)$, because $[0.6, 0.72)$ is contained in $[0.6, 1.0)$, Q_B's results is contained in Q_A's result. Thus, by interval encoding approach, the containment relationship of XPath expressions can be obtained by the computation of interval values. We will further discuss in Section 5 how to use intervals for compressed XML fragments dissemination.

4 Building QIT and Sub-index

In this section we discuss the concept of QIT, which exploits the containment relationship for XPath expressions in order to avoid server sending repeated result fragments and to support more efficient multi-query evaluation.

4.1 Query Index Tree

Definition 3 (Query Index Tree). *A Query Index Tree (QIT) is an index in a server. Suppose that there are n XPath queries Q_1, Q_2, ... Q_n. According to the containment relationship among these queries, the query tree is defined as:*

1. *The root is marked as the queried document D; all the queries are its descendants.*
2. *Each node has a set of descendants (except the leaf node) whose queries are contained by the query of current node.*
3. *Each node is represented by "(Q_{id}, begin, end, P/I)", where Q_{id} denotes the query submitted by a client; "begin" and "end" record the locations of the result fragments in the original document; "P/I" means if the result is precise or imprecise, where an imprecise result means that the result is not exact for user's query and is a larger one.*

QIT reveals the containment among queries and compressed result. The result locations, which are the values of "begin" and "end", in QIT is for queries processing at intermediate clients. As Fig. 3 shows, clients from C_A to C_I submit queries from Q_A to Q_I. According to the containment among these queries, the corresponding QIT is obtained according to Definition 3. In next section, we will describe the algorithm for building QIT.

4.2 QIT Construction in the Server

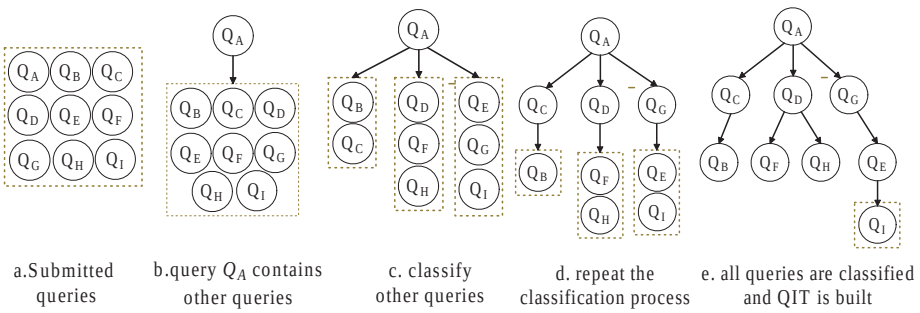

Fig. 4. Procedure of Building QIT

The main goal of QIT is to build a hierarchical structure among queries based on their containment relationship. This problem is analogous to building hierarchy classification tree, such as Yahoo taxonomy. If query Q_A contains query Q_B, Q_A is the parent of Q_B. In Fig. 4a, Q_A contains all the other queries. Thus, node Q_A is the root as shown in Fig. 4b. In Fig. 4c, queries from Q_B to Q_I are classified into three classes. Then, the larger queries which contain smaller ones

Algorithm 1 `BuildingQIT (Query Set` QS`, node` R`)`

Input: QS is a set containing simplified queries; node R is current root
Output: QIT tree

```
1: set up a new stack and add the first query into it;
2: for each query Q in QS do
3:    for each existing stack S do
4:       if Q contains S's top then
5:          S.push(Q);
6:          continue to check whether other stack tops are contained by Q and combine
            them;
7:       else if Q is contained by S's top then
8:          push Q into S and keep current top unchanged;
9:          break;
10:      end if
11:   end for
12:   if Q has not classified into existing stacks then
13:      set up a new stack and push Q into it as top;
14:      tops of current stacks become the children of R;
15:   end if
16: end for
17: for each stack S' do
18:   if S' has elements other than top then
19:      BuildingQIT (queries in this stack expect top, top of this stack);
20:   end if
21: end for
```

in each class is determined, shown in Fig. 4d. Finally, queries are organized as a tree in Fig. 4e.

We use stacks to implement this procedure where each stack represents one class, and the algorithm is described in Algorithm 1.

Initially, an empty stack is built and the first query is pushed into the stack. When a new query comes, we compare it to the tops of all current stacks. If this query is contained by the top query of a stack, it will be classified into that stack and the current stack top will remain unchanged (Steps 7–9). If this query contains the top query of a stack, we not only put it as the current stack top, but also continue to compare it with other stack, because there might exist other stack tops contained by this query. If that happens, we combine these two stacks and put this query as the new top (Steps 4–6). If there is no stack top that has containment with the current query, we have to set up a new stack for it (Steps 12–14). After all queries have been processed, each stack is a separate class. For the class that has more than one query, we recursively classify the queries and build the hierarchy according to the containment relationship (Steps 17–21). Then, the whole QIT is constructed.

The time complexity for BuildingQIT is $O(n^2)$ in the worst case, which happens when there is no containment relation among all the queries.

4.3 Sub-Index Construction for Clients

In the procedure of result delivery, the naive traditional approach in the distributed environment is to evaluate queries in the server and extract the results for each client. However, this approach is time consuming and creates heavy loading on the server. Our framework is able to reduce the server workload and bandwidth with the help of intermediate clients to transmit some compressed results according to the containment described in QIT.

In order to obtain child queries' results at intermediate(inner) clients in QIT as quickly as possible, a **sub-index** is present for each client. This sub-index is to record the result location of subsequent queries. Each result fragment sent to an intermediate client is always affiliated with the corresponding sub-index. When intermediate clients need to publish their offspring's results contained in their own result fragments, the corresponding sub-index will help the clients to locate and extract required results quickly.

Definition 4 (Sub-index). *A sub-index of query Q is the sub-tree rooted at Q in the QIT. This index includes all the result-location information of node Q's children in the QIT.*

For example, in Fig. 4e, when we send Q_D's result to client C_D, the sub-tree rooted at node Q_D will be attached. This sub-tree includes all result-location information for Q_D's children. When client C_D receives the result fragment, it scans the sub-index first, and extracts the corresponding part for clients C_F and C_H (rather than decompressing the XML fragments and evaluating queries Q_F and Q_H), and then extracts sub-indices for C_F and C_H, respectively. This explains how and why a sub-index helps efficient query processing over compressed XML fragments.

5 Multi-Query Evaluation

In this section, we discuss two issues related to query evaluation in our framework. One is how to evaluate queries using QIT over compressed XML documents. The other is how to support intermediate clients to locate results and corresponding sub-indices for its child clients. These two problems are related to how to use QIT for query processing. Here, our algorithm for query evaluation based on QIT is presented.

After compressing the XML document using XPRESS [9], the information on compression related to process queries is reserved. We use an "**Interval Table**" to keep the mapping of simple paths to the intervals. Each simple path in the document has a unique interval, which can be obtained from this Interval Table.

XPath expressions in our algorithm are considered as P, $P_1//P_2$, $P_1/*/P_2$, $P_1//P_2/*/P_3$, $P_1/*/P_2//P_3$, ..., where P_i is a simple path like $/p_{i1}/p_{i2}\ldots/p_{in}$, and p_{ij} is a label in the XML document. Thus, the P_i can be translated into "intervals" by using the Interval Table.

Algorithm 2 `TestNodes(Interval` I`, QuerySet` $QNodes$`, Boolean` B`, Structure` PS`)`

Input: I is the given interval; $QNodes$ is a set containing the query nodes which have not been tested; B indicates test all children in $QNodes$ ("true") or only complex ones ("false"); PS is the structure of I's tag.
Output: add matched children into PS.SatNodes and partly matched children into PS.WaitNodes.

```
1: for each query Q_c in QNodes do
2:   if (Q_c is a simple path) && (I is contained Q_c's interval) then
3:     add Q_c into PS.SatNodes;
4:     TestChildren( I, Q_c.children, "true", PS);
5:   else if (Q_c contains "*" or "//") && (I is contained by Q_c's first interval)
     then
6:     add (Q_c,2) into PS.WaitNodes;
7:     TestChildren( I, Q_c.children, "false", PS);
8:   end if
9: end for
```

For XPath query containing double slash, such as $P_1//P_2$, it is translated into a group of intervals. Thus, XPath query $P_1//P_2$ is separated into P_1 and P_2 by "//", and interval values of these two parts are obtained from the Interval Table. For example, "$/a/b//c/d$" will be separated into "$/a/b$" and "$/c/d$", and intervals of "$/a/b$" and "$/c/d$" will be looked up in the Interval Table

For XPath query containing wildcard, such as $P_1/*/P_2$, it is separated into P_1, "$*$", P_2, where $P_i = /p_{i1}/p_{i2} \ldots /p_{in}$ $(i = 1, 2)$. Then, we translate these three parts into their corresponding intervals. For example, "$/a/b/*/c/d$" is transformed into "$/a/b$", "$*$" and "$/c/d$".

XPath queries encoded into intervals by using Interval Table can be evaluated directly on the compressed document.

Before introducing the **QueryEvaluation** algorithm in Algorithm 3, we introduce four data structures.

UnsatNodes keeps the root of the sub-trees that cannot be matched with the current tag. Once the root cannot be matched, all its descendants cannot be matched according to the containment relationship.

WaitNodes For nodes whose queries contain "$*$" or "//", if parts of their intervals have been matched at or before this tag, WaitNodes keeps the next parts of unsatisfied intervals, and these parts will be tested with coming tags.

SatNodes keeps the query nodes that are matched with the current tag.

PathStack is a stack that keeps structures that contains the current tag, and its UnsatNodes, SatNodes and WaitNodes.

In Algorithm 3, a null PathStack is initially set up, and all child nodes of QIT's root are inserted into UnsatNodes of the root element(Steps 1–4). In processing of a start tag (encoded into an interval), WaitNodes and UnsatNodes of its parent element will be checked (Step 7). The nodes in these two structures

Algorithm 3 `QueryEvaluation(Compressed doc` Doc`, Query tree` QIT`)`

Input: Doc is the compressed XML doc; QIT contains all submitted queries
Output: QIT containing all result locations

```
 1: initiate PathStack into empty;
 2: create a path structure rootPS for root element of Doc;
 3: insert all children of QIT's root into rootPS.UnsatNodes;
 4: push rootPS into PathStack;
 5: begin to parse Doc:
 6: for each coming interval I of start tag T do
 7:    set parentPS as the top of PathStack;
 8:    create a path structure PS for T;
 9:    TestNodes(I, parentPS.UnsatNodes, "true", PS); {call Alogorithm 2}
10:    for each element (Q_t, loc) in parentPS.WaitNodes do
11:       if I is contained by the loc^th interval of Q_t then
12:          if loc^th interval is the final interval of Q_t then
13:             add Q_t into PS.SatNodes.
14:             TestNodes(I, Q_t.children, "true", PS);
15:          else
16:             add (Q_t,loc + 1) into PS.WaitNodes;
17:             TestNodes(I, Q_t.children, "false", PS);
18:          end if
19:       end if
20:    end for
21:    push PS into PathStack;
22: end for
```

will be classified into simple paths and complex ones. For each query in WaitNodes, the interval of the specific part is tested (Step 11), and when matched, its next interval of this query will be inserted into WaitNodes of the current tag (Step 16). However, it is possible that this part is the final part of the corresponding query. Then this query has been totally matched at this tag and should be inserted into SatNodes of current tag(Step 13). Besides, the child nodes of this node in QIT need to be checked recursively (Steps 14 and 17).

For UnsatNodes, Algorithm 2 is called at step 9. In Algorithm **TestNodes**, for each query, its interval can be evaluated with current tag directly, and once satisfied, the child nodes of this query in QIT are checked recursively (Steps 2–4). For complex queries that contain "$*$" or "//", the first interval will be tested. Once satisfied, the following interval is inserted into WaitNodes and its children which have complex queries are checked recursively (Steps 5–7).

6 Experiments

In this section, we describe the implementation of this prototype and test our approach on the well-known XML benchmark XMark [21], whose compression ratio by XPRESS approach is approximately 60%. We conducted all the exper-

iments on a platform with a machine of Pentium IV, 3.2 G CPU and 2 GB of RAM. This platform is used to simulate a distributed environment, where clients are simulated as threads and each client submits one query to the server. The number of clients ranges from 10 to 70.

The size of the original documents is varied from 1KB to 100MB, and the number of queries ranges from 20 to 1000. The queries used in our experiment are extracted randomly from the paths contained in the original XML document. We study these parts of our framework as follows. The first is to test the efficiency of query processing on the server side in Sections 6.1 and 6.2. The second is to verify the efficiency of reducing the server's load in Section 6.3. Finally, we compare our approach with a direct XML processing strategy in Sections 6.4 and 6.5, which demonstrate the overall benefits of our approach in terms of cost savings.

6.1 Time Performance of Building QIT

The objective in this experiment is to study the time cost of using BuildingQIT versus the synthetic data-set when running on the server side. We vary the query number and observe the CPU time changes when running BuildingQIT algorithms, in which the building time includes the cost of determining the containment relationship and constructing QIT trees for all queries.

As shown in Fig. 5, the time spent on building the query tree is roughly linearly scalable to the number of queries submitted to the server. It also indicates that the time used to build the QIT tree is negligible compared to processing XPath over a large-scale XML document when the clients increase. We also compare QIT to BloomFilter [17] on building time in order to study the efficiency of the QIT algorithm. BloomFilter is known to be highly efficient as a new tool used in XML filtering. The results confirm that the QIT building time is comparable to the building time in BloomFilter approach.

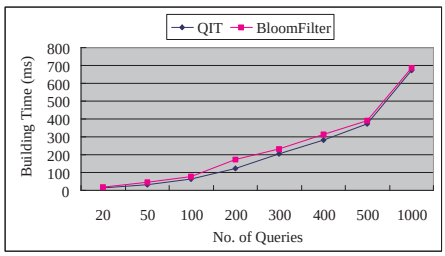

Fig. 5. QIT Vs. BloomFilter

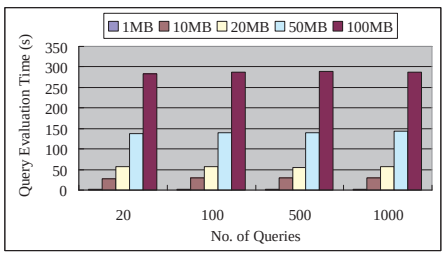

Fig. 6. Query Evaluation Performance

6.2 Performance of Query Evaluation

We now study the efficiency of query evaluation on the server side. As shown in Fig. 6, for each specified document size, the time spent on query evaluation is stable, even though the query number varies greatly from 20 to 1000 and the XML document size varies from 1MB to 100MB. This desirable feature is due to the use of QIT which is built for all queries. After translating the XPath expression into its corresponding encoded interval, the queries over the compressed document are transformed into computation of interval value according to QIT. In order to obtain the "begin" and "end" information for QIT, the compressed document is parsed only once, thus the query processing time by our approach depends mainly on the document size.

6.3 Workload of Server

The experiments in this section are executed in the simulated distributed architecture. We fix the XML document to 1MB size and limit the client number to 70. We show the efficiency of our approach in reducing the server's load during the result publication. In Fig. 7, three kinds of size ratios are used to examine the load effect of the server. The parameters are explained as follows:

Sout is the size of server's output; *Tout* is the total data size published in the network; *Uout* is the size of the uncompressed results that will be sent by traditional approach without compression; *Dout* is presented for comparison to XML filtering in Sub/Pub mode [3, 4, 17] where the server will send the whole document once it is proved to match the query. *Dout* is thus the document size multiplied by query number.

As shown in Fig. 7, when the client number is small and when there is a low containment ratio existing among queries, most of the results should be processed and sent out by the server. In this case the *Sout*/*Tout* approaches 1. When the client number is 10, *Sout*/*Uout* is close to the compression ratio of XPRESS. When the client number becomes larger, the containment ratio will also increase. As a result, the publication load of the server will be shared by the intermediate clients. Then, the server's load in the whole network is reduced. Compared with the uncompressed cases, the server's load in our system has the advantages gained by XML compression and clients' co-operative transmission.

6.4 Comparison with Simple XML Processing Strategy

In order to gain better insight into the benefits of our approach, we compare our approach with the **simple strategy**, which has neither QIT nor co-operation among clients. For each submitted query, the server directly evaluates on the original XML document. Here, we adopt SAXParser to parse the document, and then to obtain the matched results for queries. The XML document used in this experiment is fixed at 1MB.

In a distributed server-client network, performance of a system will be determined not only by the query processing time, but also by the publishing time

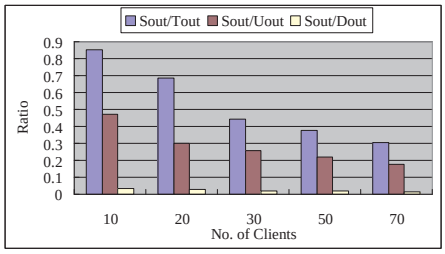

Fig. 7. Server Output Ratios

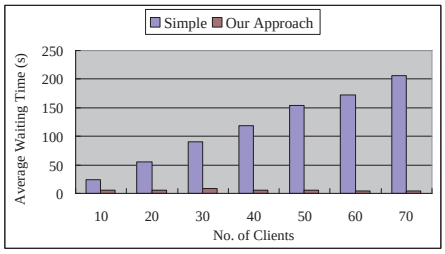

Fig. 8. Average Waiting Time

of results or the response time to the client. We use the parameter **average waiting time** given below in order to figure out the average response time for a client to receive the query result.

$$AverageWaitingTime = \frac{\sum (Tf_i - Ts)}{n} \tag{4}$$

where Tf_i is the time when the i^{th} client finishes receiving its result, Ts is the time the server begins to publish the first result, and n is the number of clients.

As shown in Fig. 8, the server evaluates queries in a linear fashion when using the simple strategy, thus the waiting time of clients increases linearly with the total number of clients which submit queries. In our approach, query results are published by both the server and intermediate clients in a multi-thread fashion. In addition, the reduced size of the results by compression in our approach enhances the overall performance.

6.5 Overall Cost Savings

We have already demonstrated how the performance of our system can be enhanced by exploiting the containment relationships existing in submitted queries. The worst case is that no containment can be used and the server has to evaluate and publish all results as the simple strategy. Whereas we still have the advantage of bandwidth savings due to the XML compression even in worst case. We now formulate the cost in the worst case W and the cost A in our approach as follows.

$$W = \sum_{i=1}^{n} (Tp_i + Tr_i) \tag{5}$$

where Tp_i and Tr_i indicate the query processing time and result publication time for the i^{th} client, respectively.

And we use A to indicate the actual condition of our approach.

$$A = Tqit + Tp + \sum_{i=1}^{n} Tr_i \tag{6}$$

where $Tqit$ is the QIT-building time of Algorithm 1, Tp is the query processing time of Algorithm 3, and Tr_i denotes the time of result publication to the i^{th} client.

We establish a parameter called the **saving ratio** as follows:

$$S = \frac{W - A}{W} \tag{7}$$

The saving ratio for querying on a 1MB-size XML document is shown in Fig. 9. When the number of clients increases, the containment ratio increases and so does the saving ratio. As intermediate clients help the server to publish the contained results in our approach, the response time of the whole network decreases. We also note in Fig. 9 an interesting phenomenon that the efficiency of query processing improves when more clients participate in asking and distributing query results.

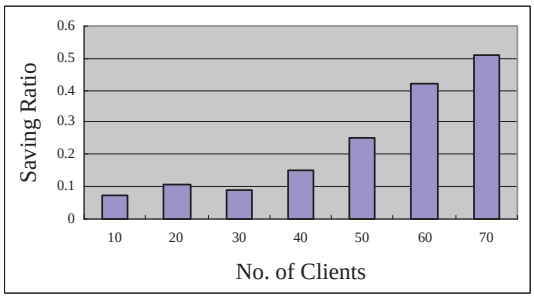

Fig. 9. Saving Ratios

7 Conclusions and Future Work

In this paper, we tackle the problem of how to process queries efficiently over a server-client architecture that consists of a large number of geographically distributed users who access a large amount of correlated XML information. We present a framework that is able to save bandwidth and process the queries efficiently. The underlying idea is to take advantage of XML compression technology and the containment relationships among queries in a co-operative client-server environment to publish XML results on the network. We also discuss some techniques to support query processing in the server side and client sides. Experimental results show that our approach is efficient in Internet-scale XML dissemination. In the future work, we will discuss dynamical maintenance of QIT and extend our scope of queries further to include more expressive XML queries such as XQuery. An orthogonal problem related to fast information dissemination is that we need to exploit the use of cache to aid the sharing of XML data that have been obtained by clients.

8 Acknowledgement

This work is partially supported by NSFC under grant No. 60496325 and 60403019, and by RGC CERG under grant No. HKUST6185/02E and HKUST6185/03E.

References

1. Tolani, P. M., Haritsa, J. R.: XGRIND: A Query-Friendly XML Compressor. In Proc. of the 18th ICDE (2002) 225–234.
2. Chen, Z., Gehrke, J., Korn, F.: Query Optimization in Compressed Database Systems. In Proc. of SIGMOD (2001) 271–282.
3. Diao, Y., Rizvi, S., Franklin., M. J.: Towards an Internet-Scale XML Dissemination Service. In Proc. of the 30th VLDB (2004) 612–623.
4. Diao, Y., Altinel, M., Franklin, M. J., et al.: Path Sharing and Predicate Evaluation for High-Performance XML Filtering. ACM Trans. Database Sys. (2003) 467–516.
5. Buneman, P., Grohe, M., Koch, C.: Path Queries on Compressed XML. In Proc. of the 29th VLDB (2003) 141–152.
6. Liefke, H., Suciu, D.: XMill: An Efficient Compressor for XML Data. In Proc. of SIGMOD (2000) 153–164.
7. Miklau, G., Suciu, D.: Containment and Equivalence for an XPath Fragment. Journal of the ACM. Vol. 51 No. 1 (2004) 2–45.
8. Neven, F., Schwentick, T.: XPath Containment in the Presence of Disjunction, DTDs and Variables. In Proc. of ICDT (2003) 315–329.
9. Min, J., Park, M., Chung, C.: XPRESS: A Queryable Compression for XML Data. In Proc. of SIGMOD (2003) 22–33.
10. Cheng, J., Ng, W.: XQzip: Querying Compressed XML Using Structural Indexing. In Proc. of EDBT (2004) 219–236.
11. Bruno, N., Gravano, L., Koudas, N., et al.: Navigation- vs. Index-Based XML Multi-Query Processing. In Proc. of the 19th ICDE (2003) 139–150.
12. Ng, W., Lam, Y. W., Wood, P., et al.: XCQ: A Queriable XML Compression System. In Proc. of WWW (2003).
13. Ng, W., Lam, Y. W., Cheng, J.: Comparative Analysis of XML Compression Technologies. To appear: World Wide Web Journal (2005).
14. Jiang, H., Lu, H., Wang, W., et al.: XR-Tree: Indexing XML Data for Efficient Structural Joins. In Proc. of ICDE (2003) 253-263.
15. Amer-Yahia, S., Koudas, N., Marian, A., et al.: Structure and Content Scoring for XML. In Proc. of VLDB (2005) 361–372.
16. Kaushik, R., Krishnamurthy, R., Naughton, J., et al.: On the integration of structure indexes and inverted list. In Proc. of SIGMOD (2004) 779–790.
17. Gong, X., Qian, W., Yan, Y., et al.: Bloom Filter-based XML Packets Filtering for Millions of Path Queries. In Proc. of ICDE (2005) 890–901.
18. XPath. http://www.w3.org/TR/xpath20/.
19. gzip. http://www.gzip.org.
20. XML. http://www.xml.com/.
21. Xmark. http://www.xml-benchmark.org/.