Efficient Multi-Query Evaluation over Compressed XML Data in a Distributed Environment

Aoying Zhou, Juzhen He, Wilfred Ng, and Xiaoling Wang

Department of Computer Science and Engineering, Fudan University, China

Email: {ayzhou, juzhenhe, wxling}@fudan.edu.cn

Department of Computer Science and Engineering, Hong Kong University of Science and Technology

Email: wilfred@cse.ust.hk

Abstract

With increasing dissemination of XML, multi-query processing has become a practical and meaningful issue to resolve. However, the verbosity of XML data causes inefficient query processing and high network bandwidth consumption. This paper addresses the problem of evaluating a heavy load of subscribed queries as a whole (or simply multi-queries) over compressed XML data in a distributed environment. We propose a holistic approach that evaluates complex queries over a compressed document and directly forwards the compressed results to clients. We first introduce a new rewriting technique, which is used to decompose and reorganize a complex query into its corresponding **S**tructure of complex **XP**ath (SXP). Then, multi-query evaluation can be performed based on the containment relationships between the queries. The containment relationships are exploited by a global data structure, **S**tructural-**Q**uery-**I**ndex **T**ree (SQIT). SQIT is an efficient structure that supports prefix sharing among the submitted queries, reserves all result locations as indexes after evaluation, and establishes a sequence of result publication. The analytical and experimental results demonstrate that the proposed approach can obtain higher query processing efficiency than traditional ones, and the bandwidth cost can be saved substantially. Thus, our approach is particularly suitable for large numbers of geographically distributed users who need to access a massive amount of correlated XML information in a cooperative distributed environment.

Index Terms: Multi-query processing, Subscribed queries, XML compression

1 Introduction

Although XML [22] has already become a de-facto standard for data representation and exchange over the Internet, the repeated tags and redundant structures give rise to the well-known data verbosity problem. The problem hinders the development of applications that involve intensive use of XML in practice, since it may lead to a substantial increase in the costs of storing, processing, and exchanging web data. In order to tackle this problem, many XML-specific compression systems, such as XMill [4], XGrind [1], XMLPPM [16] and XPress [7], have recently been proposed. Some of their methods [1, 4, 7, 8, 16] are able to support direct access to compressed documents and avoid expensive decompression in the query evaluation stage. However, these methods only supporting single-query processing are inadequate in XML filtering applications, since the method which processes queries one at a time is time consuming and is not practical for handling heavy loads of subscribed queries. More importantly, parsing compressed documents costs a lot of time.

On the other hand, query processing based on an automata-theoretic approach has been profoundly studied in XML subscription/publishing applications [2]. However, to our knowledge, filtering of compressed XML data has not been considered as an approach in literature. With the rapid increase in both the number and size of XML documents over the Internet, equipping an XML server with queriable XML compression technologies [10, 11] is a reasonable solution that deserves investigation. An XML server can be dedicated to handling documents in a compressed format. Thus processing of multi-query over compressed documents has emerged as a practical and meaningful issue to resolve.

In order to support the XML applications that involve multi-query processing and high volumes of result dissemination, we have developed succinct structures to organize queries and efficient techniques to evaluate multi-queries over large-scale compressed XML documents.

We now describe an application scenario of processing multi-queries over compressed XML documents. Assume that there are cooperative relationships among clients, as shown in Figure 1. The server maintains a compressed large-scale XML document, and clients cooperate to obtain information or news from the server. In such a scenario, it is important to adopt distributed techniques and XML compression techniques to save bandwidth in result

delivery. In our example, the server is located in London, and users from Beijing pose queries to the server. After query processing and result publishing, some results on the users' local machines may be reusable in response to subsequent queries posed from Shanghai.

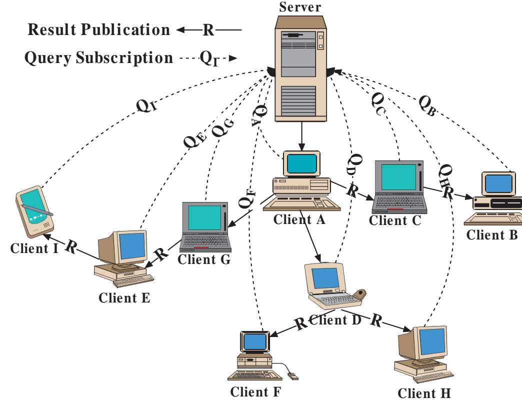

Figure 1: The Architecture of a Co-operative Framework

There are several challenges to develop an efficient approach for the above application. First, unlike usual distributed query processing, it is not feasible to process these queries one at a time, since the number of queries is extremely large. Second, XML documents are compressed in order to save storage and bandwidth. Then multi-queries need to be evaluated directly without performing full decompression of the documents. Third, with existing methods, evaluating complex queries over compressed documents is still a time-consuming process. In this paper, we not only consider the efficient evaluation of single complex queries, but also explore the containment relationships among the queries to improve the efficiency of the multi-query evaluation. Fourth, efficient result publication is essential, especially in a cooperative environment. Thus, it is necessary to develop an approach which can reduce bandwidth consumption and improves the performance of the entire network as well.

In order to address the above challenges, a novel query decomposition and organization strategy, called the **S**tructure of complex **XP**ath (SXP), is proposed as the basis of **S**tructural-**Q**uery-**I**ndex **T**ree (SQIT), which is the kernel structure in the query processing. The main contributions of this paper are highlighted as follows:

- We propose a cooperative framework for multi-query processing over compressed XML documents in order to minimize the cost of query processing and result publication.

- We develop a new query rewriting technique that organizes complex XPath queries into the **S**tructure of **XP**ath (SXP). SXP is a data structure that allows us to analyze a complex XPath query and evaluates the query on compressed documents.

- We present a novel **S**tructural **Q**uery **I**ndex **T**ree (SQIT), which can take advantage of containment relationships among queries to process queries and publish results. SQIT is a global data structure which organizes all subscribed queries as a whole. Based on SQIT, we develop efficient query evaluation and result dissemination strategies.

- We study the system maintenance issues and dynamic update algorithms of SQIT, which support inserting and deleting queries. The technique makes the whole framework robust and adaptable.

- We conduct an experimental study on the evaluation of the XPath queries on compressed XML data. Compared with existing approaches, our results show that the proposed approach is significantly more efficient. Our approach also improves system performance by reducing the size of the results and saves the consumption of network bandwidth.

The remaining part of this paper is organized as follows. Section 2 overviews the related work. Section 3 gives the preliminaries and background knowledge of our approach. Section 4 presents a decomposition model for an XPath query. The definition of SQIT and algorithms for building and maintaining SQIT are also given. Section 5 describes how we evaluate multi-queries over compressed data based on SQIT. The system architecture and the implementation are described in Section 6. Experimental study is presented in Section 7, and concluding remarks are given in Section 8.

2 RELATED WORK

The most related work is from the areas of XML query processing, XML filtering and XML compression techniques.

First, several indexing techniques have been proposed for query optimization over an XML document. For example, the structure index [9, 12, 13] offers efficient support for

path/structure queries. There are also some proposed methods [14, 15] that combine the structure index and keyword searching in XML document retrieval. For example, the integration index [15] takes the advantages of both the structure index and inverted lists. However, these methods are not applicable to processing heavy loads of subscribed queries over compressed XML documents.

Second, the techniques [2, 3, 17] developed for XML filtering are also related to our work. However, most XML filtering systems compute queries by navigating XML documents through query structures, including the prefix tree or Non-Deterministic Finite Automaton (NFA). For example, YFilter [3] employs a single NFA to represent all path queries by sharing the common prefixes of the paths.

Third, there are several emerging XML compression techniques and strategies [1, 4, 7, 8, 16], which can be classified into two categories of queriable or unqueriable compression as detailed in [11]. We only discuss the first category, since it is more relevant to this work. XGrind [1] and XPress [7] are two examples of homomorphic compressors in the first category. They both support direct querying of compressed data by retaining the document structure; XGrind uses dictionary encoding and Huffman encoding for tags and data. XPress adopts reverse arithmetic encoding for tags and diverse encoding methods for text according to the data types. The encoding technique enables XPress to achieve better compression ratios and higher query performance than XGrind. However, these methods do not support the evaluation of multi-queries over compressed documents in a co-operative framework.

3 PRELIMINARIES

In this section, the scope of XML queries, the notion of XPath containment, and the underlying idea of the *interval encode technique* [7] are introduced.

3.1 XPath Containment

An XML document is represented as an ordered labelled tree with the root node r, where a tree node corresponds to an element or a value in the XML document, as shown in Figure 2.

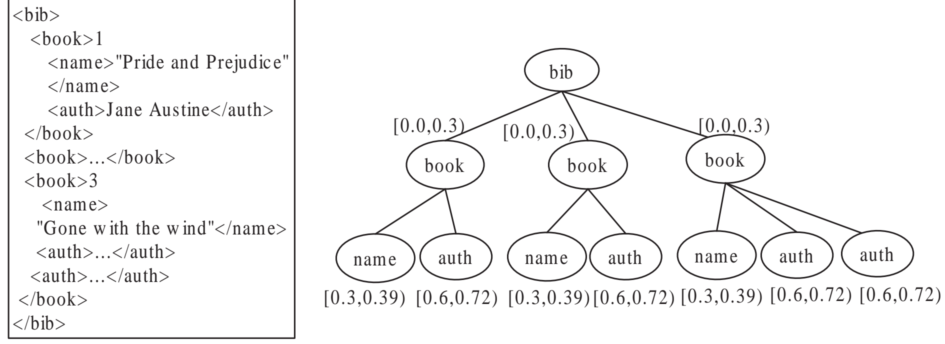

Figure 2: An XML snippet and its corresponding tree structure with encoded tags

In the subsequent discussion, only the XPath queries in $XP^{\{/,//,*,[]\}}$ are considered. The grammar of $XP^{\{/,//,*,[]\}}$ is described by the following expression:

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

where "l" is the label of the XML document, "$*$" is a wildcard, and "." denotes the current tag. Following usual notations in [21], we use "/" and "//" to mean the child and the descendant.

In this paper, we assume that all XPath queries or path expressions are $XP^{\{/,//,*,[]\}}$ queries. Furthermore, the $XP^{\{/,//,*,[]\}}$ queries containing only "/" are called *simple path queries*. Otherwise, they are called *complex path queries* as illustrated in Example 1.

Example 1 Q_1 = "$//closed_auctions[*[personID = 1]]/date[text =$ "12/15/1999"$]$" *is a complex query.*

There may exist containment relationships among different queries in $XP^{\{/,//,*\}}$, and it is possible to take advantage of these containment relationships to speed up the query evaluation and results publication. Intuitively, if query Q_A contains query Q_B in term of computing results, Q_A's result can be sent to client C_A by server, and C_A can send Q_B's result to client C_B. Thus, the server does not need to send Q_B's result to both clients, C_A and C_B. As a result, the server's load is reduced and the server's bandwidth is saved.

Definition 1 (Containment of XPath) *For two XPath queries Q_1 and Q_2, if the result of Q_1 is contained in the result of Q_2 for any given XML document, we say that Q_1 is contained by Q_2, and this fact is denoted as $Q_1 \subset Q_2$.*

The containment relationship helps to avoid some expensive and repeated evaluation of contained queries on the original document. However, the containment of the $XP^{\{/,//,*\}}$ expression is a co-NP problem. The work in [5] presents a *sufficient* but *incomplete* PTIME algorithm to compute the containment, whose underlying idea is that each XPath expression can be expressed as a *one-arity* pattern tree, and vice versa. Then, XPath expressions can be translated into *pattern trees*, and the containment is evaluated based on finding the *homomorphism relationships of the pattern trees* (or simply the *pattern homomorphisms*). Informally, a pattern homomorphism is a mapping, h, from the nodes in a pattern, p, to the nodes in another pattern tree, p' such that h is root-preserving, respects node labels and obeys edge constraints. The formal definition of h can be consulted from [5]. When there is a pattern homomorphism between two pattern trees, there also exists a containment relationship between the two XPath queries [5]. For example, Figure 3 shows a pattern homomorphism from one pattern tree p' to another pattern tree p. In this case, the XPath query XP_p, which is translated into the pattern tree p, is contained by the XPath query XP'_p, which is translated into the pattern tree p'.

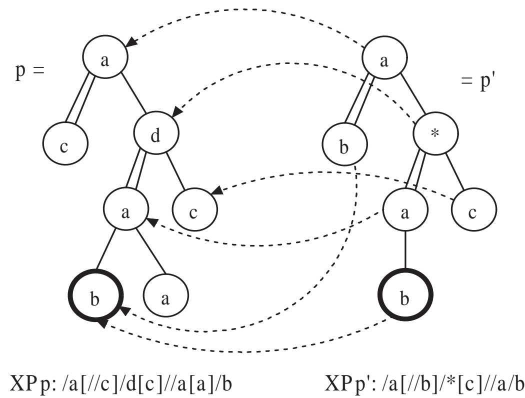

Figure 3: A pattern homomorphism from p' to p

3.2 XML Compression Technique

XML compression has the benefits of reducing the cost of both storage and result delivery on a network. We compress XML documents by using dictionary encoders for text and interval encoders for tags, which have the capability of supporting queries containing predicates. The Interval Encoding method [7, 18] is applied for encoding tags. The underlying idea is as follows: before compression, we collect the statistical information of tags by parsing the

document. Assume the name of the ith tag is $t(i)$, where i are in $(1, \ldots, m)$ and m is the number of different tags in the document. The probability of $t(i)$ is $prob_{t(i)}$ and then $t(i)$ is allocated with an initial region encode $[MIN^0_{t(i)}, MAX^0_{t(i)})$ in $[0, 1)$ based on the probability information of $t(k)$ and $1 \leq k \leq i$. Then, each tag in the document is encoded as an interval computed according to Equation 2.

$$(j = 0) \quad MIN^j_{t(i)} = \sum_{k=1}^{i-1} prob_{t(k)}, \quad MAX^j_{t(i)} = \sum_{k=1}^{i} prob_{t(k)} \tag{2}$$

For each simple path "$p_0/p_1/\cdots/p_n$", the jth tag for $j \in \{0, \ldots, n\}$ corresponds to the tag name p_j. According to Equations 3 and 4, this path is encoded as an interval computed.

$$(j > 0) \quad MIN^j_{p_j} = MIN^0_{p_j} + prob_{p_j} * MIN^{j-1}_{p_{j-1}} \tag{3}$$

$$(j > 0) \quad MAX^j_{p_j} = MIN^0_{p_j} + prob_{p_j} * MAX^{j-1}_{p_{j-1}} \tag{4}$$

As shown in Figure 2, the frequency of the tag $< book >$ is 3 and the frequency of all tags (i.e. $< book >$, $< name >$ and $< auth >$) is 10. Thus, the tag $< book >$ has the probability of 0.3. Because $< book >$ is the first element in the document, it will be allocated with an initial interval, $[min^0_{book}, max^0_{book}) = [0.0, 0.3)$, the first range fragment in [1,0). The initial intervals of $< name >$ and $< auth >$ are $[0.3, 0.6)$ and $[0.6, 1.0)$, respectively. During compression, $< book >$ is the first element and is encoded into a value in the range of $[0.0, 0.3)$. The tag $< name >$ of the path "$/book/name$" is compressed into a value in the range of $[0.3 + (0.6 - 0.3) * 0.0, 0.3 + (0.6 - 0.3) * 0.3)$, which is $[0.3, 0.39)$. And $< auth >$ with path "$/book/auth$" has the interval of $[0.6 + (1.0 - 0.6) * 0.0, 0.6 + (1.0 - 0.6) * 0.3)$, which is $[0.6, 0.72)$. All the intervals are marked with their corresponding tags in Figure 2.

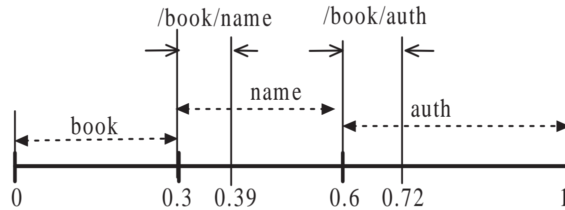

Figure 4: An Example of Interval Encoding

To evaluate queries across a compressed document, all the queries have to be translated into intervals and directly compared with the compressed tags. Based on the computing rules given by Equations 2, 3 and 4, the reverse arithmetic encoding method has a useful feature of suffix-containment, as shown in Figure 4. If the XPath expression of Q_A is the suffix of that of Q_B, then Q_A's interval contains Q_B's. Thus, to evaluate the query "$//auth$", we only need to find the tag whose value is in $[0.6, 1.0)$ (i.e. the initial value of $< auth >$). As for the query "$/book/auth$", we only need to find the tags in the range $[0.6, 0.72)$, but do not need to cache the element $< book >$ in the query evaluation.

However, the above technique is only applicable for the case of simple paths consisting of only child axes. The complex queries that contain "*", "[ ]" and "//" cannot be directly translated into intervals. Thus, we develop a new rewriting technique and some sophisticated data structures for handling complex queries, which will be presented in Section 4.1.

4 Structural Query Index Tree

In this section, some techniques are presented to process complex queries over the compressed documents. The multi-query processing strategy is based on rewriting methods that transform complex queries into efficient tree structures.

4.1 Query Rewriting

For a complex query containing "$*$", "//" or "[ ]", our approach is to split it into several simple paths or predicate expressions, called *split components*, which can be encoded into intervals. Then, the components are connected by wildcard "$*$", descendant "//" or predicate "[ ]". Each complex query may be split into several components. For example, suppose there are three components P_1, P_2 and P_3 as shown in Figure 5, we reorganize the components in order to express the queries, "$P_1/*/P_2$", "$P_1//P_2$" and "$P_1[P_3]/P_2$".

The dotted and solid directed edges in Figure 5 represent two different kinds of relationships between the components. A **primary link** specifies the components linked on the main path, which ends with the last requested element specified in an XPath expression, whereas a

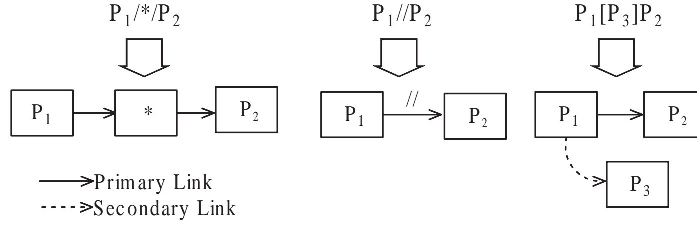

Figure 5: Query Decomposition and Reorganization

secondary link is a link between the branch or predicate component and a primary link. Components coming from links marked by "//" indicate the descendant matching, which means the "ancestor/descendant" relationship between the component and its preceding one (e.g. the query $P_1//P_2$ shown in Figure 5). Otherwise, the relationship is assumed to be "parent/child". Based on these two kinds of links and the relevant split components, a complex XPath query is translated into a data structure called SXP as defined in Definition 2.

Definition 2 (Structure of Complex XPath (SXP)) *Given an XPath query Q, the structure of the components of Q, denoted as SXP, is a tree structure defined by (V, E_j, E_n), where (i) V is a set of components that is the split component of Q; (ii) E_j is a set of primary links, which are directed edges composed of the main path except for the branches of this SXP. The ending tag on a primary link is the requested tag; and (iii) E_n is a set of secondary links, which are directed edges connecting the predicates to corresponding branch components on primary links.*

The query Q_1 given in Example 1 of Section 3.1 is decomposed into the five components of P_1 to P_5, and reorganized into an SXP as shown in Figure 6.

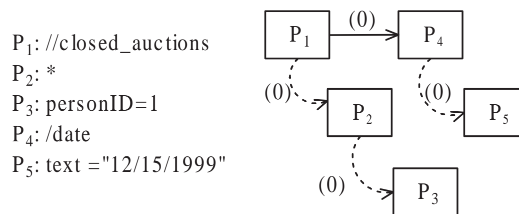

Figure 6: Components and the SXP of the complex query Q_1 in Example 1

In an SXP, each component except the root has its **preceding** component, and each component except the leaves has **subsequent** components, including components on its primary

links and secondary links. Figure 6 shows the SXP of Q_1 and its components, where a primary link running from P_1 to P_4 is the main path of Q_1. Besides, there are three secondary links: P_1 to P_2, P_2 to P_3, and P_4 to P_5. These three secondary links stand for the branch queries located at the ending element of their corresponding components. For example, the secondary link from P_1 to P_2 indicates the fact that there is a predicate expression for the branch element "closed_auctions" of P_1.

Now, we present the algorithm of transforming a complex query into SXP in Algorithm 1, where the input query Q is first decomposed into several components. The components are then reorganized into an SXP as an output. According to the level of branches, the algorithm parses the complex query and splits it recursively. In this algorithm, P_i stands for the path of the ith component appearing in the main path of the input query. We use Cxp_i to indicate the ith component of P_i. For each component on the main path, we check whether it has secondary links (lines 7–9) and then attach each component with secondary links to the resulting SXP (line 10). When applied to Q_1, we first find P_1, then its secondary link P_2 (with P_3) is attached. The final result of the SXP is illustrated in Figure 6.

Algorithm 1: TransSXP (Query Q)

Input: An XPath query Q
Output: The SXP of Q

```
1  begin
2      Initiate t_sxp as the result SXP
3      Set m_end pointing to the end of the primary link of t_sxp
4      Partition the main path of Q into a set of components, {P_1, ..., P_n}, by "*" and "//"
5      for each component P_i do
6          Create a Cxp_i for P_i
7          if P_i has a secondary link "[Pred_i]" then
8              Set SXP min_i := TransSXP(Pred_i)
9              Add min_i into the secondary link of Cxp_i
10         Add Cxp_i into t_sxp
11         Change m_end into Cxp_i
12     Return t_sxp
13 end
```

4.2 Structural Query Index Tree

We now introduce the strategy of processing multi-queries over compressed data based on SXPs. The essence of the strategy is that all submitted queries are composed into a Struc-

tural Query Index Tree (SQIT) by exploiting the containment relationship and shared prefixes among SXPs. This strategy helps minimize the processing time and publication load. The definition of the SQIT is now given as follows:

Definition 3 (Structural Query Index Tree (SQIT)) *Given a set of subscribed XPath queries* S_Q= $\{Q_1, Q_2, \ldots, Q_n\}$*, the Structural Query Index Tree (SQIT) of* S_Q *is defined by the triplet* (V_Q, E, R)*, where each component is given as follows.*

- V_Q *is a finite set of query nodes, in which each node corresponds to a unique query in* S_Q*.* E *is the set of edges representing the parent-child relationship in the tree.* R *is the (virtual) root of the tree. We will use the terms "query" and "query node" interchangeably in our subsequent discussion.*

- *Each query node is defined by (*Q_{cid}*,* SXP_Q*,* $begin[]$*,* $end[]$*), where "*Q_{cid}*" is the corresponding identity of the client, "*SXP_Q*" is the SXP of the original query* Q*, and "*$begin[]$*" and "*$end[]$*" keep the beginning and ending positions of the fragments of the query result in the compressed document.*

- *All query nodes constitute the descendant set of* R*.* E *is the set of edges that represent the containment relationship between the nodes in* V_Q*.*

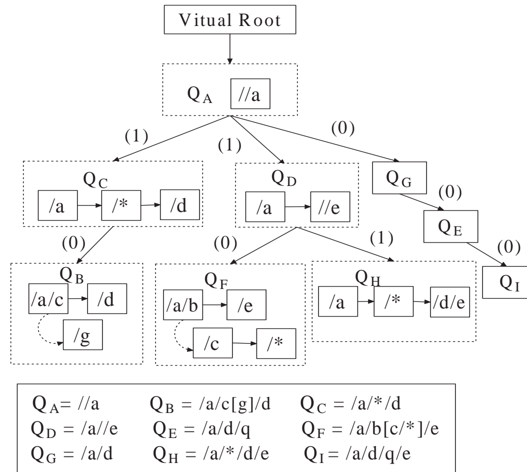

Figure 7: An Example of the Structural Query Index Tree (SQIT)

Example 2 (SQIT) *Figure 7 shows an example of SQIT. The set of queries,* $S_Q = \{Q_A, Q_B, \ldots, Q_I\}$ *listed in the box are submitted queries which are organized into the tree as shown. It can be seen that when* Q_B *is contained by* Q_A*, the node for* Q_B *is a descendant node of* Q_A*.*

In a nutshell, SQIT is a *global* data structure that organizes all subscribed queries as a whole, whereas SXP provides a *local* data structure to deal with a single complex XPath query as discussed in Section 4.1. SQIT reveals the containment relationship of query results among the ancestor and descendant nodes. Thus, if an XML fragment as a result cannot be satisfied by any node, there is no need to check its children or descendant queries in the subtrees of SQIT. Consequently, the whole search space for all queries is greatly reduced.

4.3 Sharing in SQIT

With containment relationships, SQIT and SXP provide a method to explore shared **prefix components** among queries. As shown in Figure 7, when a tag "$< a >$" comes, the first component "$/a$" of Q_D can be satisfied, as can the second child Q_H. Thus, the first components of Q_F and Q_H should be evaluated together; otherwise, the path information of each element in Q_D's result will be lost after it is satisfied.

To avoid repeatedly comparing tag "$< a >$" with the first components of Q_D and Q_H, or recursively comparing it to another component at even deeper level, we assign a **sharing index** to the directed edge from the branch node to the SXP child, once Q_H has a child also starting with a "/a" component. Returning to Example 2, the sharing index "(1)" on the edge from Q_D to Q_H means that Q_H shares the first component with its parent. For Q_F, the interval of "$/a/b$" is not equal to that of "$/a$", thus the sharing index of Q_D and Q_F is "(0)".

4.4 Building SQIT

SQIT is generated by comparing the containment relationship among queries in a recursive manner. First, complex queries are decomposed and reorganized into SXPs by using Algorithm 1, since complex queries cannot be directly translated and evaluated over the compressed document. Then, a stack structure is used to store the query nodes of a branch, and the nodes in a stack are recursively classified based on the containment relationship. The details of the procedure are shown in Algorithm 2.

Initially, an empty stack is built and the first query is pushed into the stack (line 2). When a new query arrives, it is compared to the queries on the top of all current stacks. If this query is

contained by the top query of a stack, it is pushed into that stack after the original top is popped out, and the original stack top is pushed back and the stack top remains unchanged (lines 11–14). If this query contains the top query of a stack, it should be put on the top of the stack. The query is also compared to other stacks, since there may exist other stack whose top queries are contained by this query. In this case, these two stacks are combined and the query is assigned as the new top (lines 7–10). If there is no stack whose top query has the containment relationship with the new arriving one, we have to set up a new stack for it (lines 15–16). After all queries have been processed, each stack stands for a separate class. For those classes that have more than one query, we recursively classify the queries and build the hierarchy according to the containment relationships (lines 19–22) until the whole SQIT is constructed.

A query may be contained by two or more branches of SQIT as a result of containment computing. Thus, the containment relationship of queries may be a graph instead of a tree. For example, the query expression, $Q_H =$ "$/a/*/d/e$", is contained by both query expressions, $Q_C =$ "$/a/*/d$" and $Q_D =$ "$/a//e$". A basic approach to tackle this ambiguity is to adopt some heuristic rules, such as choosing the branch which has fewer nodes (i.e. Q_D in the example). The reason is that when the size of a branch, say the number of its nodes, is small, the amount of queries to be evaluated should be small. An alternative is to classify this query into the class represented by the first generated stack which contains the query as our algorithm does.

4.5 SQIT Maintenance

In practice, subscribed queries may not arrive or leave in bunches but change one or a few at a time. In this section, the maintenance of SQIT during inserting or deleting subscribed queries is discussed.

As illustrated in Algorithm 3, when inserting a new query Q_t into SQIT, we proceed to check if there exists containment relationship between Q_t and the child nodes under the root R of SQIT (line 3). If we find a node Q_i such that $Q_t \subset Q_i$, the scope of comparison is extended into the children of Q_i (lines 4–6). If $Q_i \subset Q_t$, it replaces Q_i and makes Q_i its child, and then the siblings of Q_i are checked, until no containment can be found (lines 9–14). But

Algorithm 2: BuildingSQIT(Query Set QS, node R)

Input: QS is a set of queries; node R is the current root of SQIT tree
Output: SQIT tree

```
1  begin
2      Set up a new stack and add the first query into it;
3      for each query Q ∈ QS do
4          if Q is a complex query then
5              Decompose and reorganize Q into SXP by Algorithm 1
6          for each existing stack S do
7              if S.top ⊂ Q then
8                  S.push(Q)
9                  Continue to check whether other stack tops are contained by Q
10                 Combine all such stacks into one and push Q in the combined stack as its top
11             else
12                 Q ⊂ S.top
13                 Push Q into S but keep current top unchanged by poping out and pushing in
                   the original S.top
14                 break
15         if Q has not classified into existing stacks then
16             Set up a new stack and push Q into it
17     Calculate the prefix sharing index between tops of current stacks and R
18     Set the tops of current stacks as the children of R, mark sharing index on the edges
19     for each stack S' do
20         if S' has other elements except the top T' then
21             Set QS' as the query set of elements except T'
22             BuildingSQIT (QS', T')
23     return R
24 end
```

if none of the children contains Q_i, it is set as a new child of the current parent (lines 13–14). After traversing SQIT, if there is no node containing Q_t, Q_t will be added as a new child of R (lines 15–16).

Algorithm 4 illustrates the process of deleting a node, Q_t, from a given SQIT. We first check if there exists containment relation between its children and its following siblings. The reason of conducting this checking is that a query is always assigned to the class standing by the first branch that contains the query. It is possible that other branches also contain Q_t. Therefore, its child subtrees are inserted into its following siblings (lines 5–8). If the insertion fails, its child subtrees are then inserted at its position as the children of Q_t's parent (line 10). If Q_t is a leaf of the SQIT, we delete it accordingly.

Algorithm 3: SQIT_Insert (SQIT root R, Query Q_t))

Input: R is the root node of SQIT; Q_t is the query waiting to be inserted
Output: SQIT containing Q_t

1 **begin**
2   int $insert_loc = -1$
3   **for** *each child* Q_i *of* R **do**
4   **if** $Q_t \subset Q_i$ **then**
5   SQIT_insert(Q_i, Q_t)
6   **break**
7   **else**
8   **if** $Q_i \subset Q_t$ **then**
9   **if** $insert_loc = -1$ **then**
10   Replace Q_i with Q_t and put Q_i as Q_t's child
11                    $insert_loc = i$
12   **else**
13   Remove Q_i from the children of R
14   Add Q_i as a new child of Q_t
15   **if** *no containment between* Q_t *and the children of* R **then**
16   Add Q_t as a new child of R
17   return R
18 **end**

Algorithm 4: SQIT_Delete(SQIT root R, Query Q_t)

Input: SQIT and the node waiting to be deleted
Output: SQIT without Q_t

1 **begin**
2   Let Q_t's parent be Q_p
3   **for** *each* Q_c *in* Q_t*'s children* **do**
4   **for** *each* Q_j *in* Q_t*'s following siblings* **do**
5   **if** $Q_c \subset Q_j$ **then**
6   SQIT_Insert (Q_j, Q_c)
7   Keep the subtree of Q_c at new location
8   **break**
9   **if** *none of* Q_t*'s following sibling contains* Q_c **then**
10   Add Q_c as a new child of Q_p
11   return R
12 **end**

5 Multi-Query Evaluation

With SQIT and SXP, the subscribed queries can be evaluated across compressed XML data. We first discuss the evaluation of a single query over the compressed document and then present the approach for processing multi-queries.

5.1 Single Query Evaluation over SXP

Given the SXP of an XPath query, the evaluation of the query over a compressed document is done by traversing the SXP, where each component is translated into an interval and evaluated over the compressed document directly.

During this procedure, the link types among the components of an SXP should also be considered. If the link type is a descendant one, say $C_P//C_S$, then once the preceding component C_P is satisfied by a coming compressed XML element E, the subsequent component C_S should be compared with all descendant elements in the XML fragment rooted at E. If the link type is a child one, say $C_P/*/C_S$, when the preceding component C_P is satisfied by the element E, the subsequent component, such as C_S, need to be compared only the sub-elements in the second level of E's fragment.

In addition to the relationship between SXP and its preceding component, the type of the link where the component comes from should also be considered for its evaluation. When a component C_M comes after a secondary link, which means that it is a branch component, the goal is to check if there is any match in the fragment where its preceding component C_P is satisfied. If C_M is linked to a primary link of SXP in the result fragment of its preceding C_P, all C_M's matching results should be cached until C_P's fragment ends. To sum up, for an unsatisfied component, the locations and times of its evaluation depend on its link to its preceding component and the link type it comes from.

5.2 Multi-Query Evaluation over SQIT

We now explain how SQIT is used to support multi-query processing. We first introduce the evaluation rules and then use an example to illustrate the whole process.

5.2.1 Evaluation Rules

Let $n_i \in V_Q$ be a node corresponding to the query $q_i \in S_Q$ in an SQIT and T_i be a subtree rooted at n_i. We use "SXP children" to indicate n_i's children whose queries are complex ones and have been translated into their corresponding SXPs. During the query evaluation, if q_i is a complex query, it can be **partially satisfied** when parts of its components are satisfied, or

fully satisfied when the whole query is satisfied. For example, the query “//a” is fully satisfied by the tag “$< a >$”, but the query “/a[c]/b” is only partially satisfied by this tag. In general, for an element E in the compressed document, we consider the following rules applied for the two cases:

Case (1): if q_i is partially satisfied, it is possible that some of the SXP children of n_i can also be partially satisfied at E, thus the SXP children of n_i should be checked. However, there is no need to check the children with a simple path when q_i is not fully matched, since q_i contains all the query of its children.

Case (2): if q_i is fully satisfied at E, all children of n_i should be checked simultaneously.

If any child of q_i is partially or fully satisfied, the above rules are recursively applied to this child node according to Cases (1) and (2).

In Figure 7, Q_A is fully satisfied by the tag “ $< a >$ ”, and its children Q_C and Q_D are partially satisfied. Thus, Q_B, Q_F and Q_H, as the SXP children, should also be checked.

5.2.2 Evaluation Strategy

During the process of query evaluation, the compressed document is parsed as a SAX stream. For each coming compressed XML tag, each query in the SQIT has one and only one of the following three states: satisfied, unsatisfied and partially satisfied. Thus, three respective data structures are designed for each tag T of the compressed document. First, the structure **UnsatNodes** keeps the roots of the subtrees in SQIT whose nodes have not been satisfied after the tag T comes. Second, the structure **WaitCXPs** keeps the subsequent components of the components satisfied at T, but their query is partially satisfied by T. Third, the structure **SatNodes** keeps those nodes that are satisfied when parsing the tag T. From now on, these three structures are collectively called the **path structure** for a compressed XML tag.

For a coming compressed XML tag T, if a query in SQIT cannot be satisfied, there is no need to check any of its descendants. We only keep the unsatisfied state of those nodes (i.e. the roots of the unsatisfied states) in the stack. Because there exists a containment relationship between the ancestor query and the descendant queries, once the root of this subtree and T are not matched, all the descendant nodes and T are not matched either. As shown in Figure

8, when the tag " $< a >$ " comes, the subtree rooted at Q_G is an unsatisfied subtree, and therefore Q_G will be inserted into UnsatNodes of the tag " $< a >$ ".

There also exist some nodes that are partially satisfied in Figure 8, such as Q_C when the tag " $< a >$ " comes into the system. These queries in the figure are complex ones and are expressed as SXPs. For each partially satisfied SXP, we keep the state of "waiting to be compared to the subsequent components of the satisfied components" in WaitCXPs.

The query evaluation algorithm based on the path structure is shown in Algorithm 5. Initially, for the root element of the queried document, the path structure is constructed to contain all the children of the root query node in SQIT (lines 2–4). When a new compressed tag T comes, suppose the parent tag of T is P, each query node Q_u in P's UnsatNodes and each CXP in P's WaitCXPs should be checked (lines 5–19).

Algorithm 5: QueryEvaluation(Compressed doc Doc, SQIT $Sqit$)

Input: Doc is the compressed XML document, $Sqit$ is the SQIT containing all subscribed queries, *PathStack* is the stack for holding path structures of coming tags

Output: The stack containing the results for the queries in $Sqit$

1 **begin**
2 Create a path structure PS_r for root of Doc
3 Insert children of $Sqit$'s root into PS_r.UnsatNodes
4 Push PS_r into $PathStack$
5 **for** *each coming tag* T *with interval* I_T **do**
6 Set PS_T as the path structure of T
7 Set PS_p as the top of $PathStack$
8 TestChildren(I_T,PS_p.UnsatNodes,$false$,PS_T)
9 **for** *each* Cxp_w *in* PS_p*.WaitCXPs* **do**
10 **if** $I_T \subset Interval_{Cxp_w}$ **then**
11 **if** Cxp_w*'s query node* Q_w *is fully satisfied* **then**
12 Add Q_w into PS_T.SatNodes
13 TestChildren(I_T,Q_w's children, $true$, PS_T)
14 **else**
15 Add children of Cxp_w into PS_T's WaitCXPs
16 TestWaitCXPs(I_T, Q_t, Cxp_w's $No.$, PS_T)
17 **if** *(*I_T *is not contained by* $Interval_{Cxp_w}$*) or (*Cxp_w*'s type is "Descendant")* **then**
18 Add Cxp_w into PS_T.WaitCXPs
19 Push PS_T into $PathStack$
20 Return $PS_T.SatNodes$
21 **end**

For any node Q_u in P's UnsatNodes, Algorithm 6 is employed to test the node against

Algorithm 6: TestChildren(Interval I, QueryNodeSet $QNodes$, 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 the query is a simple one ($true$) or complex one ($false$); PS is the structure of I's tag

Output: PS.SatNodes with matched children and PS.WaitNodes with partially matched children.

```
1  begin
2      for each query node Q_c in QNodes do
3          if Q_c's query is a simple path then
4              if B = true  &&  I ⊂ Interval_{Q_c} then
5                  Add Q_c into PS.SatNodes
6                  TestChildren( I, Q_c.children, true,PS)
7              else
8                  Add Q_c into PS.UnsatNodes
9          else
10             Set Cxp_f as the first CXP of Q_c
11             if I ⊂ Interval_{Cxp_f} then
12                 Add CXP children of Cxp_f into PS.WaitCXPs
13                 TestWaitCXPs( I,Q_c, 1, PS)
14             else
15                 Add Q_c into PS.UnsatNodes
16 end
```

the current tag. If Q_u's query is a simple path, its interval value will be compared with the coming interval value directly in order to check whether or not they are matched (lines 3–8). Otherwise, Q_u's path expression is transformed into an SXP, and its root component should be checked against the tag T (lines 10–15). If it is satisfied, the subsequent components of the root component become waiting components for the next coming tag by inserting them into the WaitCXPs (line 12–13). Besides, if Q_u's root component is fully satisfied, all SXP children of Q_u in SQIT are checked, which is implemented by the procedure TestWaitCXPs in Algorithm 7. If Q_u's root CXP and T are not matched, it is added into UnsatNodes of T (line 15).

Algorithm 7 is designed for testing the SXP children of the input query node over the current interval I. A sharing index, S_i is to exploit the sharing prefixes between any two queries and it is used to skip some evaluation in the algorithm. Once the number of the current component is smaller than S_i, this child component is skipped. However, if the number is equal to S_i, the subsequent components of the current component are added into WaitCXPs (lines 9–

Algorithm 7: TestWaitCXPs(Interval I, QueryNode Q, Integer N_{cxp}, Structure PS)

Input: I is the interval of the current compared tag;
Q is the query node whose children with complex queries should be evaluated;
N_{cxp} is the number of the considered component of Q;
PS is the structure of the current tag.
Output: PS.WaitCXPs with added CXP children

```
1  begin
2    for the i^th SXP child node Q_i of Q do
3      Set S_i as the sharing index from Q to Q_i
4      Set Cxp_f as the first CXP of Q_i
5      if S_i > N_cxp then
6        TestWaitCXPs(I,Q_i,N_cxp,PS)
7      else
8        if S_i = N_cxp then
9          Add CXP children of the N_cxp th CXP of Q_i into PS.WaitCXPs
10         TestWaitCXPs(I,Q_i,N_cxp,PS)
11       else
12         if (S_i = 0)   &&   (I ⊂ Cxp_f Interval) then
13           Add CXP children of Cxp_f into PS.WaitCXPs
14           TestWaitCXPs(I,Q_i,1,PS)
15 end
```

10) in order to evaluate this part with the next tags. For those SXP children that share nothing with its parent query, all the first components are evaluated (lines 13–14).

The following example helps illustrate the multi-query evaluation with SQIT.

Example 3 (Evaluation Procedure) *In Figure 7, Q_A, Q_C, Q_D, Q_B, Q_F and Q_H are complex queries submitted by their respective clients. These queries are inserted into the SQIT and are transformed into SXPs. The components of a query are denoted by Q_{ij}, where $i \in \{A, B, \ldots, H\}$, and j is an index to represent the identity of a component in the query. As the evaluation procedures illustrated in Figure 8, Q_A, as a child of the root, is initially put into UnsatNodes of the root element in the document. When the compressed tag "$< a >$" comes, in UnsatNodes of the root element, Q_A is checked and found to be qualified, thus its child nodes in the SQIT should also be checked. Clearly, the first components of Q_C and Q_D can be satisfied by "$< a >$", the SXP children of Q_C and Q_D and the root component of Q_H are checked if they are satisfied by "$< a >$". For the satisfied components, the subsequent components (Q_{C2}, Q_{D2} and Q_{H2}) are inserted into the WaitCXPs of the current tag "$< a >$". Although Q_G is a simple path, there is no containment with an interval of "$< a >$" after*

comparing the encoded interval value. Thus, Q_G is inserted into UnsatNodes and we can skip checking its children. Similar actions are conducted for other new coming tags until the tag "$< d >$" comes. Consequently, Q_C, as an SXP, is fully satisfied and inserted into SatNodes, whereas its child node Q_B, as an unsatisfied SXP, is inserted into UnsatNodes.

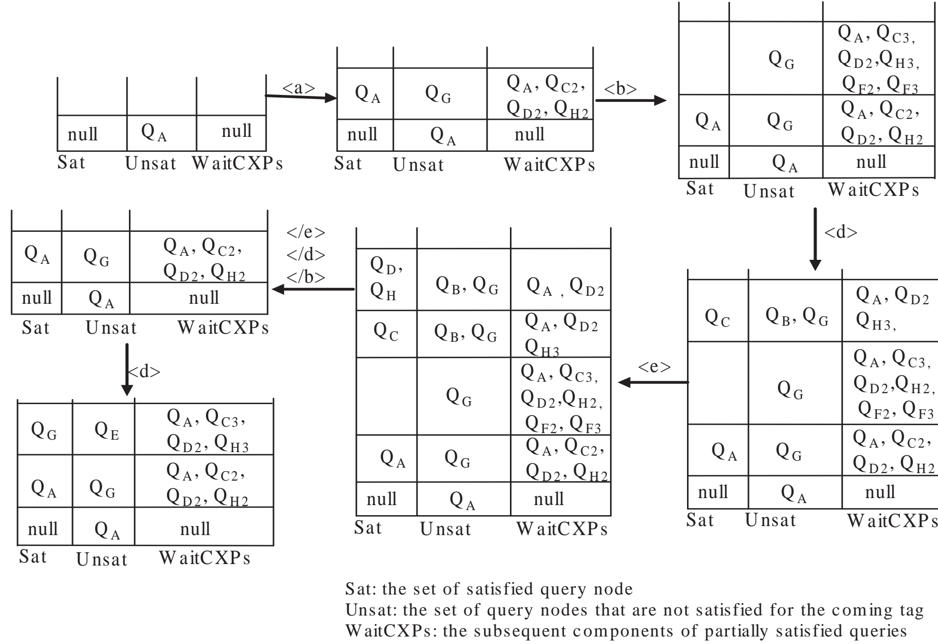

Figure 8: An Example of Query Evaluation Using the Path Structure

In our multi-query evaluation algorithm, the worst case happens when SQIT is a flat tree, where no sharing and containment can be utilised. In this case, our approach reduces to single query evaluation. However, the case of a flat SQIT rarely happens. For example, the average level of SQIT is from 4 to 6, and clients' queries are closed in our datasets.

6 System Architecture

In this section, the basic components of the system, the procedure of query processing in the whole network and the issues concerning the system maintenance are discussed. There are two main phases in the process of query evaluation. The first phase is to build the containment relationship among clients. The second phase is to publish compressed XML fragments to clients.

6.1 System Design

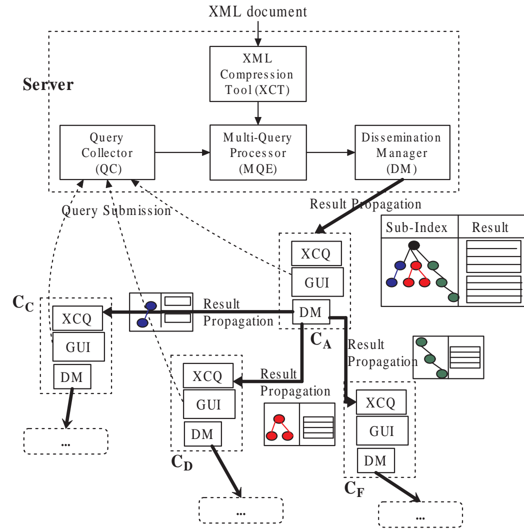

Figure 9: The System Architecture for Multi-Query Processing

Our prototype is a cooperative system built over a network of clients. The server takes the responsibility to collect queries, process them and forward parts of results to its corresponding clients. As Figure 9 shows, the server consists of four main components as follows:

XML Compression Tool (XCT). This tool facilitates the compression and storage of XML documents. XML documents are compressed using an interval encoder for XML tag and a dictionary compressor for compressing elements' contents in this system.

Query Collector (QC). This component collects queries subscribed by clients, decomposes and reorganizes them into SXPs, computes the containment relationship among queries and builds SQIT. The detailed algorithms are discussed in Section 4.

Multi-Query Evaluator (MQE). This is the most important component in the server, which evaluates queries over certain documents according to SQIT. The evaluation techniques are explained in Section 5.

Dissemination Manager (DM). This component obtains the results and intercepts subindexes for those queries at the child nodes of the SQIT root. Then, for each client of these nodes, DM propagates the result fragments and the subindex. This component is also reserved for intermediate clients to share their results with other clients.

A client in the cooperative environment of our framework first subscribes its queries to the server. The query evaluation component, called **X**ML **C**ompression **Q**uery processor (XCQ), obtains the query result from the packages forwarded by the client's parent node and conducts children's queries according to the subindexes. Figure 9 shows those propagated result packages in broad-brush arrows. Then, according to the received subindex and results, a client shares its compressed results with its corresponding child clients through the DM.

6.2 System Initialization

First, each client subscribes an XPath query to the server. Then a SQIT, which serves as a global and kernel structure, is built at the server side. The SQIT reveals the containment relationships among subscribed queries and supports the procedures of evaluation and dissemination. After building the SQIT in the initialization phrase, the server disseminates the SQIT and its related information in all clients in order to build the relationships among them, and then the whole cooperative network is established.

We now discuss how the cooperative clients share their result fragments with others. The subscribed query of each client is regarded as a node in SQIT. Those clients who share XML fragments with others are intermediate clients. For example, in Figure 7 the clients subscribing the queries Q_A, Q_C, Q_D, Q_G and Q_E are intermediate clients. An intermediate client is responsible for executing its children queries to obtain the result and publish the result fragment. Thus, each intermediate client has the information of its child clients and their result locations. We impose a subindex as a local structure on each intermediate client which contains all children's information. The subindex for the client C_i is the subtree of the SQIT rooted at the node of C_i's query. Once a client subscribes a query that corresponds to a branch node in the SQIT, the corresponding subindex is forwarded to this client, and the subindex is set as a local structure to improve the query performance of the whole system.

6.3 System Maintenance

Our system is built upon a dynamic cooperative network, where each client in our system is able to subscribe new queries, cancel subscribed queries or leave the network at any time. However, there are still some technical issues concerning system maintenance.

New Coming Client. Let C_i be a new client entering into the network. C_i sends its query Q_i directly to the server. The server inserts Q_i into the SQIT, evaluates this query in the result fragments of its parent according to the SQIT, and obtains a new index for the subscribed query. After the query insertion, the server sends a message to those clients, whose local SQIT, as a part of the SQIT, is influenced by Q_i, and the corresponding parent of C_i is also required to send results to C_i in future processing.

New Subscribed Query. If an existing client C_j subscribes a new query Q_j, local checking of C_j should be conducted as follows.

- If C_j has a local SQIT, it checks whether Q_i is contained in the local SQIT. If this is the case, there is no need to send this information to the server. The query can be evaluated over the local XML fragments directly.

- If the local SQIT does not contain Q_j, this query is then sent to the server. Then Q_j is inserted into the SQIT at the server, according to the SQIT maintenance algorithms presented in Section 4.5. The remaining procedure is the same as the last step in handling new client insertion, which requires an update of influenced local SQITs. Then, the query result is propagated from C_j's corresponding parent clients.

Unsubscribed Query. If a client in the network unsubscribes a query, the client sends an "unsubscribe request" message to the server. Algorithm 4 is then run at the server side to delete the query. Finally, the server publishes a new SQIT to those influenced clients.

Membership Testing. Our network is a server-centered and cooperative one. Thus, all the refreshing work of SQIT should be supervised by the server. In order to test if a client still exists or not, the server adopts a "ping-pong" message strategy [19], which is commonly used in a distributed environment. A broadcasted "ping" message is sent to each client regularly. If the server gets a "pong" message from a client, it means that the client exists or is online; otherwise, the client is no longer being a member of this system. Then the server updates the SQIT and sends updated subindexes to those influenced clients.

7 Experiments

The algorithms discussed in the previous sections are implemented and extensive experiments are carried out on synthetic and real data sets as shown in Table 1. All the experiments were

conducted on a PC with Pentium IV 3.2 GHz CPU and 2 GB of RAM.

Table 1: Characters of Data Sets

Data Set	XMark	NITF
Data type	Auction data	New industry text format
DTD characteristic	Simple structure; Few attributes; Large element content; Deep nested levels	Complex structure; Many attributes; Few element content; Deep nested levels.
Data size	1MB to 50MB	1KB to 1MB
Compression ratio	47.3%	65.4%
Query size	500 to 1000	1000 to 3000
Example query	/site/regions/*/item [@id='0']/description//keyword	/nitf/body/*[tagline /pronounce//q[@id=8]]//bibliography

7.1 Experimental Setting

The synthetic data is a set of XML documents generated by using IBM's XMLGenerator [25], and the real data is a commonly used XML benchmark dataset called XMark [23]. The synthetic documents generated with NITF (New Industry Text Format) DTD [24] are attribute-abundant, text-few and structure-complex. XMark documents have relatively simple structures with heavy textual content and deeply nested levels for some elements.

Both synthetic and real queries are used in our experiments. The real query set is obtained from the XMark query benchmark, which provides about 40 queries for auction data. The synthetic query set is obtained from YFilter XPath generator, which generates queries according to the parameters of query lengths, wildcard probabilities, predicate probabilities and other options. In our experiment, we set a group of default parameters to generate different queries, in which the probabilities of "$*$", "//", "[]" and "[[...]]" are 0.1, 0.1, 0.05 and 0.05, respectively. We also generate 100 to 3000 queries with respect to XMark DTD and NITF DTD. In our system, the average compression ratio of XMark and NITF XML documents are 47.3% and 65.4%, respectively. The features of these two kinds documents are summarized in Table 1.

The objectives of our experiments are threefold. The first is to study the performance of building SQIT and processing queries in our system. The results are presented in Sections 7.2 and 7.3. The second is to evaluate the advantage of processing queries directly on compressed data, our approach is compared with SAXON and YFilter in Sections 7.4 and 7.5.

The comparison is based on the time spent on evaluation and result compression for SAXON and YFilter, and the time spent on building SQIT and the query evaluation in our approach. The third is to test the efficiency of the whole network in terms of output ratio, waiting time and network profit. The results are presented in Section 7.6.

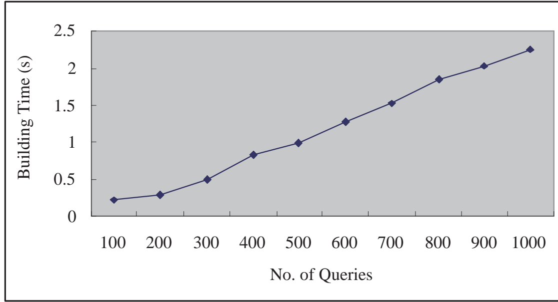

Figure 10: Building Time of SQIT

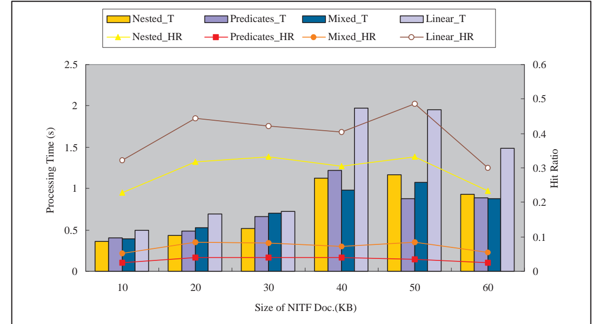

Figure 11: Performance on Four Query Types

7.2 The Building Performance of SQIT

This experiment is designed to study the performance of Algorithm 2 with respect to the synthetic queries. We first generate queries by YFilter path generator with default parameters and then test the CPU cost of the algorithm with different numbers of queries. The building time includes the cost of checking the containment relationship between subscribed queries, transforming complex queries into SXP and constructing the structure of SQIT trees.

The results are given in Figure 10. The time for building SQIT is roughly linearly scalable to the number of subscribed queries. In addition, the building time is less than 2.5 seconds, even with 1000 queries. Compared with the time needed for processing XPath queries over a large-scale XML document, this once-off construction time is reasonable.

7.3 Performance of Query Evaluation

This experiment is designed to study the performance of the query evaluation with respect to different data sets and a variety of queries. The performance of the query evaluation on both NITF and XMark documents are tested against a range of significant parameters of document size, query type, query number and query length.

7.3.1 Query Evaluation over NITF

Over the NITF document, four groups of queries generated by the YFilter Query Generator are used. Each group has 1000 queries. The query type of the first group is called "Linear", which contains only "$*$" and "//". The second one is called "Nested", which contains "$*$", "//" and nested "[ ]". The third one is called "Predicates", which contains "$*$", "//" and predicate expressions like "$[@id = 1]$". The fourth one is called "Mixed", which consists of the previous three query types.

When evaluating queries over compressed documents, the branch queries require lots of intermediate results to be cached. Descendant components will be evaluated repeatedly in each fragment that is satisfied by its preceding components. Predicates require not only structure matching, but also the value validation, which needs checking the text compressed by the dictionary encoder.

Figure 11 presents the Hit-Ratio (HR), determined by the fraction, $\frac{the\ number\ of\ matched\ queries}{total\ number\ of\ queries}$, and the query processing time T (bar charts) on four groups of queries. Note that the queries are generated by YFilter query generator, and some of these queries do not match input documents. Thus, HR has impact on T and the evaluation efficiency will be influenced by the document size, query type, and hit-ratio. "Linear" queries have the highest HR (denoted as $Linear_HR$) and the processing time is the longest one because the descendant components in linear queries are checked repeatedly. Although "Mixed" queries are more complex than "Predicates" and "Nested" queries, the HR value for "Mixed" queries (denoted as $Mixed_HR$) is not much higher than its counterparts. In a nutshell, the time spent on evaluating 1000 mixed queries with a 10% HR is limited into 1.5 seconds for all documents.

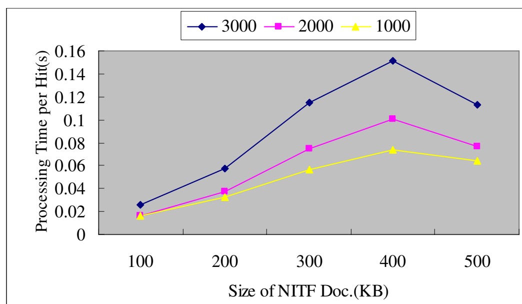

Figure 12: Performance on Query Number

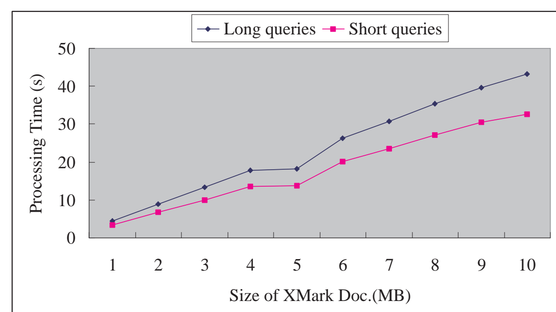

Figure 13: Performance on Query Length

In Figure 12, we use 1000, 2000, and 3000 complex queries to evaluate NITF documents ranging from $100KB$ to $500KB$. In this experiment, we use the metric, $PTH = \frac{Processing\ Time}{Hit\ Number}$, to observe the impact of query length and document size on performance of our SQIT approach. The PTH indicates the time spent on matching one query out of different groups and across different-size documents. As Figure 12 shows, the PTH value for the group of 3000 queries is the highest, and that for the group of 2000 queries takes the second place. When the document size doubles, the PTH is roughly twice of the original one. However, with a double number of queries, the time spent on evaluating a query is far less than twice of the processing time. The efficiency of our approach is affected less by the query number than by the document size. This finding is encouraging, since our main goal is to process a heavy load of subscribed queries as a whole on compressed documents.

7.3.2 Query Evaluation over XMark

This experiment is to study the impact of query length of an XMark document, which has a larger size than an NITF document. The group with four tags on average is classified as "Short" queries, and the one with ten tags is classified as "Long" queries. Each group has 1000 queries. The short queries get results from the document and the long queries get a 90% match.

Figure 13 illustrates the impact of query length on query processing. The processing time of the long queries is comparable to the processing time of the short queries. Our approach is found to be scalable to document size for both short and long queries. The time for evaluating 1000 queries of the two groups on a 10MB compressed document is less than 50 seconds in real time, which is acceptable since the system returns the exact and compressed results for the queries.

7.4 Comparison with SAXON

SAXON [26] is one of the commonly used XPath processors. We now compare the processing time of our approach with the processing time of SAXON. The processing time of our

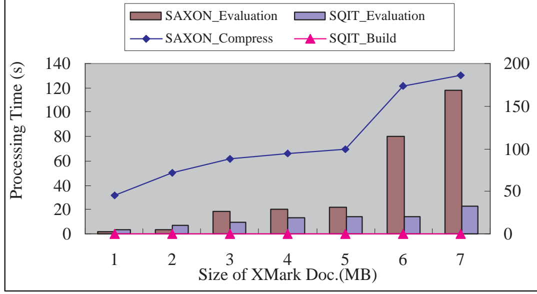

Figure 14: SQIT Vs. SAXON

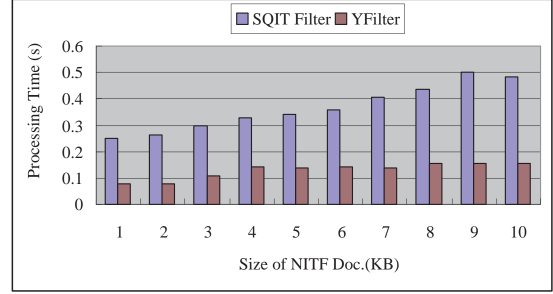

Figure 15: SQIT Vs. YFilter

approach, T_{SQIT}, can be defined as follows:

$$T_{SQIT} = T_{SQIT_Building} + T_{SQIT_Evaluation}. \tag{5}$$

We also test 100 complex mixed queries and the following equation,

$$T_{SAXON} = T_{SAXON_Evaluation} + T_{Result_Compression}, \tag{6}$$

is used to compute the processing time of SAXON. Then the output results obtained from SAXON and SQIT are both under compression.

The size of the XMark documents ranges from $1MB$ to $7MB$. As shown in Figure 14, SQIT has a stable performance on both query evaluation and its building cost is very small for various document sizes. For the processing time of small size documents, SQIT is comparable to SAXON, since our approach needs to traverse the SQIT tree. However, the system greatly outperforms SAXON when the document size increases.

7.5 Comparison with YFilter

Our system can also be used in content-based XML filtering applications. This experiment shows the performance of our approach when comparing it with YFilter [2]. In the experiment, SQIT is regarded as a filter to find the first matched tag for each query. Here, the size of the NITF data ranges from $1KB$ to $10KB$, which is small but common in XML filtering applications. Queries are limited to 1000 complex queries generated by the YFilter path generator.

As Figure 15 shows, our approach takes nearly twice as much time as YFilter. The reason is that, when finding matched tags for each query, the non-leaf nodes should be evaluated

repeatedly until all of its children are satisfied. On the other hand, our approach can find the matched elements, which is a fragment of the whole document, whereas YFilter only gives the answer “Yes” or “No” for each document. The results obtained from SQIT are small and the bandwidth is reduced. This indicates another advantage of our approach.

Our approach is shown to be a feasible approach for content-based filtering in this experiment. Although our approach consumes some time overhead, the benefits gained from bandwidth savings is significant. Taking both processing time and result publishing time into consideration, SQIT is comparable to YFilter on time. Note that we can obtain a compressed query result, whereas the size of the documents published by YFilter is nearly 60 times larger than the size of compressed results obtained from our approach.

7.6 Profit of the Whole Network

In this subsection, we compare our approach with an approach that does not make use of SQIT to explore the containment relationship, where the server takes the responsibility to forward each query result to its corresponding clients. The comparison focuses on the following three aspects, the workload of the server, the average waiting time for clients, and the cost savings for the whole network. The experiments are conducted in a simulated distributed architecture, where the client number is set to 1000, and each client submits one query.

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

7.6.1 Workload of Server

We study the efficiency of our approach in reducing the server’s workload during result publication. Figure 16 shows the size of the output for the two situations. The one marked “server” is the size of the server’s output with our approach. As intermediate clients share the publication load of the server, their output is very limited and stable even with 1000 queries. But for the simple strategy, which is marked as “Total”, the server has to forward all the query results

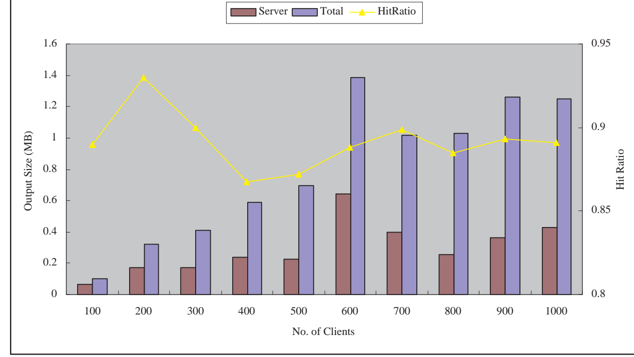

Figure 16: Output Comparison

directly to the clients. In this case, even if the results are compressed data, their total output size is still very large.

As shown in Figure 16, when the client number is small and there is a low containment ratio existing among queries, most of the results should be published by the server. In this case, the advantage of our approach is not so obvious as that of having more queries. When the client number increases, the containment ratio also increases. As a result, the publication load of the server will be shared by the intermediate clients. Thus, the server's load in the whole network is reduced.

7.6.2 Comparison with a Simple Strategy

In a distributed server-client network, the performance of a system is determined not only by the query processing time, but also by the publishing time of the results or the response time to the client. The parameter **average waiting time** given below is used to determine the average response time for a client to receive the query result.

$$Average\ Waiting\ Time = \frac{\sum_{i=1}^{n}(T_{f_i} - T_s)}{n}, \quad (7)$$

where T_{f_i} is the time when the ith client finishes receiving its result, T_s is the time the server begins to publish the first result, and n is the number of clients.

As shown in Figure 17, the server evaluates queries in a linear fashion when using the simple strategy. Thus, the waiting time for 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. As a result, the average waiting time for

clients is stable when the query number becomes larger. In addition, the reduced size of the results by compression in our approach enhances the overall performance.

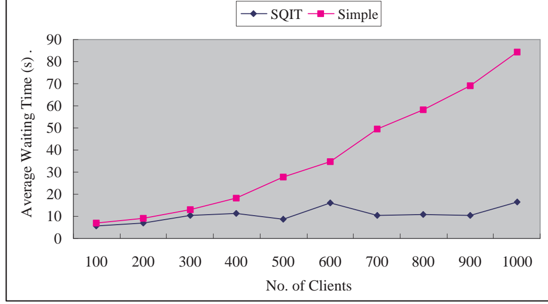

Figure 17: Average Waiting Time for Clients

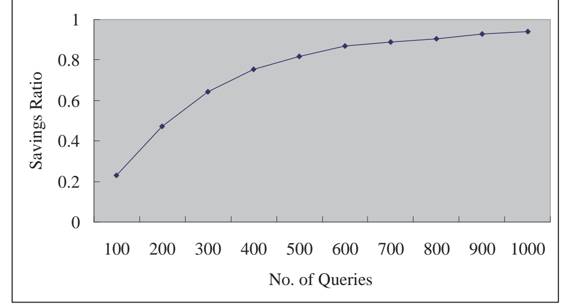

Figure 18: Savings Ratio of Processing Cost

7.6.3 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. However, under this case, we still have the advantage of bandwidth savings due to the reduced size of the compressed answer. The cost in the worst case, W, can be calculated as follows:

$$W = \sum_{i=1}^{n} (Tp_i + Tr_i), \tag{8}$$

where Tp_i and Tr_i indicate the query processing time and result publication time for the ith client, respectively.

The cost of our approach can be computed as follows:

$$A = T_{SQIT} + Tp + \sum_{i=1}^{n} Tr_i, \tag{9}$$

where T_{SQIT} is the building time of the SQIT specified in Algorithm 2, Tp is the query processing time of the system as specified in Algorithm 5, and Tr_i denotes the time of result publication to the ith client.

We establish a parameter, the **savings ratio**, given as follows:

$$Savings\ Ratio = \frac{W - A}{W}. \tag{10}$$

The savings ratio for querying on the XMark document in our approach is shown in Figure 18. When the number of clients increases, the containment ratio also increases, as does the savings ratio. Because intermediate clients help the server to publish the contained results in our approach, the response time of the whole network decreases. Figure 18 shows an interesting phenomenon that the efficiency of the query processing even improves as more clients participate in querying and distributing query results.

8 Conclusions

In this paper, a holistic approach is presented for processing a heavy load of subscribed queries efficiently over compressed XML documents in a cooperative distributed environment. Efficient query evaluation techniques are proposed to process multi-queries in a dynamic environment. The underlying idea is to utilize XML compression technologies to process the queries as a whole directly across the compressed data, rather than process the queries with a demand-driven decompression. From a technical perspective, we exploit containment relationships between queries to publish compressed XML results in order to reduce the bandwidth. We develop two efficient data structures, SXP and SQIT, to handle XPath queries as well as dynamic updates of subscribed queries. The dynamic maintenance issues of queries and clients have also been taken into account in our system as well. From the empirical perspective, we have conducted comprehensive experiments, which show that our approach is efficient for compressed XML date dissemination. There are two interesting extensions to this work. One is the scope of queries could be further extended to include more expressive XML queries such as XPath/XQuery Full-Text [20]. Another orthogonal problem related to fast information dissemination is to exploit the use of caches to offer support for sharing of compressed XML data that have been obtained with clients.

References

[1] Tolani, P. M., Haritsa, J. R.: XGRIND: A Query-Friendly XML Compressor. In Proc. of the 18th ICDE (2002) 225–234.

[2] Diao, Y., Rizvi, S., Franklin., M. J.: Towards an Internet-Scale XML Dissemination Service. In Proc. of the 30th VLDB (2004) 612–623.

[3] 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.

[4] Liefke, H., Suciu, D.: XMill: An Efficient Compressor for XML Data. In Proc. of SIGMOD (2000) 153–164.

[5] Miklau, G., Suciu, D.: Containment and Equivalence for an XPath Fragment. Journal of the ACM. Vol. 51 No. 1 (2004) 2–45.

[6] Neven, F., Schwentick, T.: XPath Containment in the Presence of Disjunction, DTDs and Variables. In Proc. of ICDT (2003) 315–329.

[7] Min, J., Park, M., Chung, C.: XPRESS: A Queryable Compression for XML Data. In Proc. of SIGMOD (2003) 22–33.

[8] Cheng, J., Ng, W.: XQzip: Querying Compressed XML Using Structural Indexing. In Proc. of EDBT (2004) 219–236.

[9] 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.

[10] Ng, W., Lam, Y. W., Wood, P., et al.: XCQ: A Queriable XML Compression System. In Proc. of WWW (2003).

[11] Ng, W., Lam, Y. W., Cheng, J.: Comparative Analysis of XML Compression Technologies. World Wide Web Journal (2006), Vol. 9, No. 1, 5-33.

[12] Qun, C., Lim, A., Win, K.: D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. SIGMOD (2003) 134–144.

[13] Jiang, H., Lu, H., Wang, W., et al.: XR-Tree: Indexing XML Data for Efficient Structural Joins. In Proc. of ICDE (2003) 253-263.

[14] Amer-Yahia, S., Koudas, N., Marian, A., et al.: Structure and Content Scoring for XML. In Proc. of VLDB (2005) 361–372.

[15] Kaushik, R., Krishnamurthy, R., Naughton, J., et al.: On the integration of structure indexes and inverted list. In Proc. of SIGMOD (2004) 779–790.

[16] James, C.: Compressing XML with multiplexed hierarchical models. In Proc. of IEEE Data Compression Conference (2001) 163–172.

[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] Chen, Y., B. D., Susan, Zheng, Y.: BLAS: An Efficient XPath Processing System. In Proc. of SIGMOD (2004) 47-58.

[19] Ripeanu, M., Foster, I., Iamnitchi, A., Mapping the Gnutella Network. IEEE Internet Computing Journal(2002, Vol.6, No.1) 50-57

[20] XQuery 1.0 and XPath 2.0 Full-Text. http://www.w3.org/TR/2005/WD-xquery-full-text-20051103.

[21] XPath. http://www.w3.org/TR/xpath20/.

[22] XML. http://www.xml.com/.

[23] XMark. http://www.xml-benchmark.org.

[24] NITF. http://www.nitf.org/index.php.

[25] IBM XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[26] SAXON. http://saxon.sourceforge.net.