Efficient Query Processing for Large XML Data in Distributed Environments

Hiroto Kurita†, Kenji Hatano‡, Jun Miyazaki†, and Shunsuke Uemura†

† Graduate School of Information Science, Nara Institute of Science and Technology
Keihanna Science City, Ikoma, Nara 630–0192, Japan
‡ Faculty of Culture and Information Science, Doshisha University
1-3 Tatara-Miyakodani, Kyotanabe, Kyoto 610-0394, Japan
{hiroto-k|miyazaki|uemura}@is.naist.jp, khatano@mail.doshisha.ac.jp

Abstract

We propose an efficient distributed query processing method for large XML data by partitioning and distributing XML data to multiple computation nodes. There are several steps involved in this method; however, we focused particularly on XML data partitioning and dynamic relocation of partitioned XML data in our research. Since the efficiency of query processing depends on both XML data size and its structure, these factors should be considered when XML data is partitioned. Each partitioned XML data is distributed to computation nodes so that the CPU load can be balanced. In addition, it is important to take account of the query workload among each of the computation nodes because it is closely related to the query processing cost in distributed environments. In case of load skew among computation nodes, partitioned XML data should be relocated to balance the CPU load. Thus, we implemented an algorithm for relocating partitioned XML data based on the CPU load of query processing. From our experiments, we found that there is a performance advantage in our approach for executing distributed query processing of large XML data.

1 Introduction

Extensible Markup Language (XML) [8] is a recommended standard of the World Wide Web Consortium (W3C) that is increasingly being used for various forms of data exchanges between differing data sources. XML applications have been growing, and in recent years, uses for large scale data such as a digital library system have been expanding.

In normal applications, we use Document Object Model (DOM) [7] or Simple API for XML (SAX)[1] for processing XML data. However, DOM and SAX are not appropriate for processing of large sized XML data since the processing cost of these methods increases greatly with the size of XML data. Instead, an efficient query processing approach tailored for large XML data is required. Moreover, processing large XML data on single computation node is confronted with the problem of limited CPU power as well as limited capacity of hard disk drives and size of physical memory. Thus, we propose an efficient large-scale XML data query processing system that solves these problems by partitioning and distributing XML data to multiple computation nodes. Distributed processing has several advantages, including parallel processing, load balancing, scalability and redundancy.

The primary considerations when constructing a distributed query processing system for large XML data are:

1. Data partitioning
   In order to achieve an efficient query processing for large XML data, the XML data must be partitioned evenly based upon data partition size. Moreover, XML data is a tree structure, so that the XML data has to be partitioned preserving the integrity of the tree structure. Therefore, the data partitioning algorithm must consider the data size as well as its structure.

2. Data distribution
   Similar to data partitioning, it is necessary to consider data size and data structure when distributing the partitioned data. The query processing cost as well as storage cost of XML data is dependent on the data size, so that we should evenly distribute partitioned data among computation nodes. If the partitioned data can be distributed in such a way, query processing cost is distributed among the multiple computation nodes. Therefore, it is required to propose a technique for balancing the processing costs among the computation nodes. This is also related to dynamic data relocation, which is mentioned below.

3. Distributed query processing
   The query management is handled by a control node.

[1] http://www.saxproject.org/

21st International Conference on Advanced Networking and Applications(AINA'07)
0-7695-2846-5/07 $20.00 © 2007 IEEE

The control node manages mapping of path information of decomposed XML data. When queries arrive at the control node, they are analyzed and dispatched to appropriate computation nodes. After issuing the queries to computation nodes, it is responsible for compiling the results from each of the computation nodes. It is vital for the control node to efficiently and correctly issue each query to computation nodes.

4. Dynamic data relocation
   As mentioned in explanation of data distribution, it is important to efficiently make use of computation nodes as much as possible. Thus, partitioned and distributed data should be relocated to balance the query processing cost when the frequencies of queries change or the query processing cost is unbalanced. It is important to consider which distributed data should be relocated to balance the query processing cost of computation nodes after the changes.

Considering distributed query processing of large XML data on digital library systems, the design of the XML data can be straightforward, i.e., the records of each book information should be closely located. In addition, users usually issue simple XPath queries to find information related to one book. In this case, structural join is rarely required to get the search results of the queries if the subtrees of the book data are not decomposed.

In this paper, we develop an XML query processing system for large XML data considering four above-mentioned points in distributed environments for digital library systems. In addition, we compare and evaluate three XML data partitioning algorithms, and find the most effective one.

The rest of this paper is organized as follows: In Section 2, we describe related work. In Section 3, we explain our approach for constructing a distributed query processing system for large XML data on digital library systems. In Section 4, we show the comparative results of our experiments, and we conclude our proposal in the final section.

2 Related Work

There have been many studies of distributed systems for relational databases; however, few of them consider distributed XML databases. Data decomposition and dynamic data relocation of distributed relational databases are classic problems, so there are already various proposed methods to solve them [3, 4]. Since relational databases are arranged in tuples, data partitioning and relocation is relatively easy. However, due to the XML tree structure, data partitioning and relocation becomes more complicated.

Bramer et al. and Andrade et al. proposed efficient distributed processing for XML databases [1, 2]. The key idea of these systems is based on a structure summary called Repository Guide, which is similar to Strong Data Guides, in order to decompose XML data. However, this technique requires three indexes to maintain partitioned data. Kido et al. also proposed a method for partitioning XML data on a PC cluster after mapping to relational tables [5]. In this approach, XML data mapped to tables is decomposed based on the frequencies of the paths accessed by XPath queries.

These partitioning approaches were essentially achieved by decomposing XML data into vertical and horizontal fragmentations, which originally come from [2]. However, these studies did not address the problem of skewed query processing and storage costs. Although our approach is basically based on vertical fragmentation, it attempts to address the skew problem through data partitioning, distribution, and dynamic data relocation.

3 Distributed Query Processing System

Our distributed query processing system for XML data is as follows. To create an efficient distributed query system for large XML data, we need to consider the following: data partitioning, data distribution, distributed query processing and dynamic data relocation based on both the query processing and storage costs. We now explain each of these parts in detail.

3.1 Summary of Proposed Distributed Query Processing System

Our proposed system is shown in Figure 1. First, we partition the large XML data and evenly distribute the data partitions to multiple data nodes, called Query Processors (QPs). Queries are sent from users to the control node, called the Query Controller (QC). The QC analyzes the queries to determine which QP nodes to send the query. The QC then forwards the query to the appropriate QP nodes. QP nodes execute the query via an XML query processing engine. Query results from each QP are then sent back to QC, where the results are integrated and finally returned to users.

3.2 Data Partitioning

As for partitioning XML data, there are various strategies. For example, considering only the data size, and considering only data structure.

It is important for distributed query processing to balance the storage and query processing costs on each QP. That is, we need to consider balanced data size and load of query processing when distributing data. The storage cost depends on the data size, while the query processing cost is associated with both the data size and its data structure. Therefore, we propose a data partitioning method which takes the data size as well as the data structure into account.

The partitioning method we propose decomposes XML data into several fragments, so that each fragment can be the same size while preserving the original data structure as much as possible. The reason we took such a strategy is to balance both storage and CPU costs. Considering storage

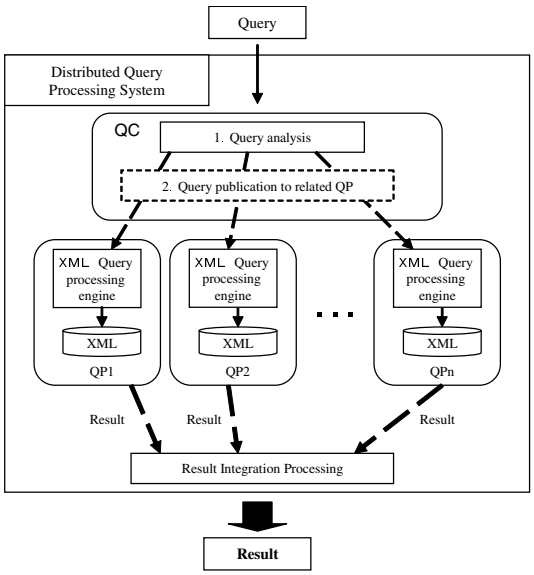

Figure 1. Overview of Our System

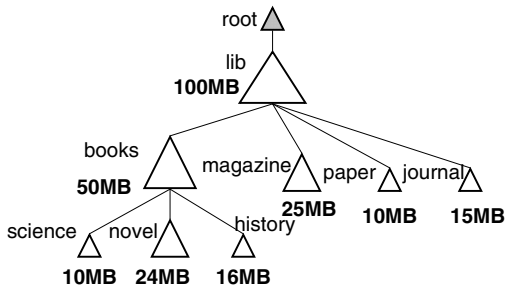

Figure 2. An example of XML data

cost, the size of each partitioned XML data should be similar. The size of XML data is associated with CPU cost as well. Whether DOM or SAX model is used to parse XML data, CPU cost and data size correlate to each other. Considering the structure of XML data, on the other hand, each partitioned XML data should also keep in its original data structure, because query processing for XML data is greatly based on the structure. If XML data is decomposed without considering the structure, more CPU processing would be required to reconstruct the original structure. This is the key difference from distributed relational databases.

Here, we describe our partitioning algorithm briefly. Let the size of original XML data be M, and the number of fragments be N. An ideal data size after partitioning is expressed as $\alpha = \frac{M}{N}$. Since it is difficult to decompose XML data into the same sized fragments, we introduce a permissible margin of fragments ϵ, such that $\alpha(1-\epsilon) \leq L \leq \alpha(1+\epsilon)$, where L denotes the permissible size range of fragments.

1. Parse XML data once, and get the number of child nodes and the size of the subtree in each node recursively.

2. For a child node n_c of a node n in the XML data, calculate the size of n_c, which is denoted as S_{n_c}. If the size S_{n_c} falls in the permissible size range L, then we retain the subtrees whose parent node is n_c, as a whole.

3. If S_{n_c} is larger than L, go to step 2 on a child node of n_c, which is denoted as n_g.

4. If S_{n_c} is smaller than L, we combine n_c and its siblings one by one until the sum of them falls in the permissible size range L. A new subtree is then created by combining the nodes.

5. If the size of the new subtree exceeds the range of L, the node which causes oversize is skipped.

6. If the new subtree does not reach α, the procedure moves back up to the parent node and goes to step 4 on the parent node, whose size should now be below the range of L.

7. Repeat step 1 through 6 on the entirety of XML data.

In order to demonstrate our partitioning algorithm, an example of XML data is shown in Figure 2. In Figure 2, the numbers accompanying each node represent the sizes of the respective subtrees. We apply our algorithm to decompose the XML data into four fragments. Since we set $M = 100, N = 4, \alpha = 25$, and $\epsilon = 0.1$ as their initial values, the size range of each fragment is $22.5 \leq L \leq 27.5$.

In the XML data in Figure 2, there are four child nodes of node `lib`: `books`, `magazine`, `paper`, and `journal`. We process the nodes starting from node `books`. Since the size of node `books` is larger than L, we process the child nodes of `books`. There are three child nodes of node `books`: `science`, `novel`, and `history`. We evaluate these child nodes in this order. Since the size of node `science` is smaller than L, the node is added to a temporary tree. The temporary tree now contains the node `science` and its size is 10MB. Next, we move to node `novel`. Since the size of `novel` falls in L, we retain the subtree below node `novel` as whole. Since the size of node `books` is smaller than L, it is added to the temporary tree. The temporary tree now contains nodes `science` and `history` and its size becomes 26MB. Its new size now falls within the range of L, and thus, we store this subtree by combining the nodes. The rest of the tree is processed in the same way. The four data fragments generated by this algorithm are shown in Figure 3. Data identifiers `DataIDs` are assigned to each node. Lastly, the node is linked up to the root by adding all the missing nodes from the root of the subtree.

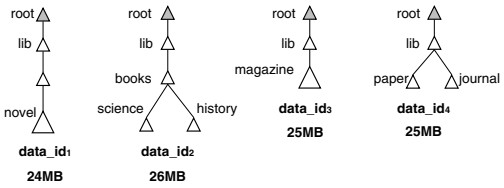

Figure 3. Decomposed XML data

3.3 Data Distribution

When decomposed XML data are distributed to QPs, query workload of each QP is not predictable. Therefore, we initially distribute partitioned XML data to each QP without any consideration. Our XML data partitioning algorithm mentioned in Section 3.2 is based on both data size and its structure, so that we arrange the same number of partitioned XML data to each QP.

After distributing partitioned XML data, we create a mapping table (referred to as *datamap* in this paper) on QC. *Datamap* has three attributes: `key` is the original XPath expression from root node to a node, `DataID` represents the identifier of partitioned XML data, and `QPID` is QP's identifier. With these three attributes, QC can recognize which QPs should handle which queries.

3.4 Distributed Query Processing

When users issue queries to QC, QC analyzes the queries to determine which QP contains results related the queries. As we described in the previous section, QC can find both `DataID`s and `QPID`s by using `key`s extracted from the queries. Therefore, QC can send the queries to QPs which have the fragments containing the results to the queries, and then, each QP returns the answers to QC. Finally, QC integrates all of partial results from QPs and returns a final result to the users. Note that for the sake of simplicity, we perform data integration on the QC itself. However, it is not essential. It can be processed on any idle computation node when QC is busy.

3.5 Dynamic Data Relocation

So far, each QP is well balanced from the storage cost point of view. However, CPU cost on each QP cannot be balanced when considering storage cost because query processing cost on each QP is dependent on query workloads. If this skew occurs, efficient query processing cannot be carried out in distributed environments. This is because certain QPs with heavy query processing costs will become bottlenecks, leading to performance degradation. To cope with this problem, we apply a dynamic relocation scheme for partitioned XML data based on the query workloads when the query processing cost of QPs is unbalanced.

Relocation of fragments is done as follows:

1. QC manages query processing times of each QP during some period and determines QP_L and QP_S, where QP_L is the QP with the longest query processing time, and QP_S is the QP with the shortest query processing time.

2. QC also calculates d_L and d_S, which are partitioned XML data with the longest and shortest query processing time on QP_L and QP_S, respectively.

3. QC asks the QPs to exchange d_L and d_S with each other. Exchanging d_L and d_S allows the query processing time on QP_L to be reduced when the query processing time of QP_S is shorter. During the relocation process, the entries of d_L and d_S in *datamap* are locked to avoid dispatching queries to wrong QPs.

4. QC updates the *datamap* in response to relocation of fragments.

Repeating the above process, total query processing time is reduced because query processing times of each QP are balanced.

Even with this relocation algorithm, there are cases in which a query can be dispatched to the wrong QP which does not have the corresponding fragment any longer because it has already moved to another QP. In this case, such a query is sent back to QC and re-dispatched to the appropriate QP.

4 Experimental Evaluation

The purpose of this evaluation is to show the advantage of dynamic data relocation by using our XML partitioning algorithm in distributed environments. Our test platform is composed of five nodes (CPU: AMD Opteron 2.4GHz $\times$ 2, main memory: 16GB) connected to a gigabit network. One of the nodes is used as QC, and the others are QPs. The software was developed on the Sun J2SE Development Kit (version 1.5.06). We used XSQ (version 1.0) [6] as the XPath processing engine on QPs.

In our experiments, we used 105.2MB of XML data generated by `xmlgen`[2] and 24 queries as shown in Table 1. The XML data is decomposed into several XML fragments based on our proposed algorithm as well as two other ones. These fragments are distributed to QPs evenly with respect to their size. We assume that our XML data partitioning algorithm is used to decompose large XML data on digital library systems, and we do not consider queries which require structural join techniques.

[2] `xmlgen` is provided by XMark project.

Table 1. Our Query Set

	Query		Query
Q1	//date	Q13	/site/regions/namerica/item[@id="item10000"]//text
Q2	//name	Q14	/site/regions/namerica/item[@id="item13000"]//text
Q3	//seller	Q15	/site/regions/namerica/item[@id="item15000"]//text
Q4	/site/regions//item	Q16	/site/regions/samerica/item[quantity="1"]//mail
Q5	/site/regions/namerica/item	Q17	/site/categories/category/name
Q6	/site/open_auctions/open_auction/initial	Q18	/site/people/person[@id="person12000"]
Q7	/site/regions/europe/item	Q19	/site/open_auctions/open_auction[@id="open_auction35"]
Q8	/site/people/person/name	Q20	/site/open_auctions/open_auction[@id="open_auction3000"]
Q9	/site/regions/asia/item[quantity="1"]	Q21	/site/open_auctions/open_auction[@id="open_auction5600"]
Q10	/site/regions/australia/item[payment="Creditcard"]	Q22	/site/open_auctions/open_auction[@id="open_auction8300"]
Q11	/site/regions/europe/item[@id="item5000"]	Q23	/site/closed_auctions/closed_auction/price
Q12	/site/regions/europe/item[@id="item7500"]	Q24	/site/closed_auctions/closed_auction//text

4.1 Evaluation of Data Relocation

We firstly evaluate the efficiency of dynamic data relocation in our system. In order to evaluate the efficiency of dynamic data relocation, we measure query processing times of our system in two cases: with and without dynamic data relocation. We use data partitioning algorithm described in Section 3.2 to decompose original XML data. In this experiment, the original XML data is decomposed into 16 XML fragments, and each of four XML fragments is distributed to the same QP. This is because it helps to balance query processing times among QPs at the data relocation process compared with one fragment per one QP policy. This policy is similar to the fine grain bucket approach for skew handling in parallel hash joins [3]. In order to measure the query processing time, we issue total 2,400 queries chosen from the query set shown in Table 1. Also, data relocation is processed every 600 seconds in our system. We execute the same experiments ten times and evaluate the results using the average CPU time.

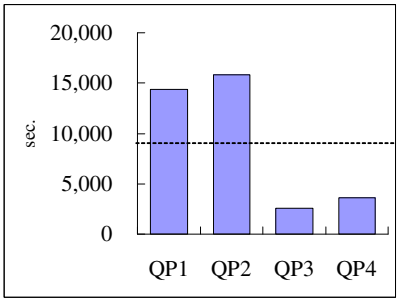

Figure 4. Query processing time on each QP without data relocation

Figures 4 and 5 show that the data relocation helps to balance query processing times among QPs. We also measured the data relocation cost, and found that it takes only less than one second for each relocation. Therefore, the overhead can be ignored. Consequently, total CPU time of our system with data relocation becomes faster than without data relocation, even if it includes the relocation overhead (see Table 2).

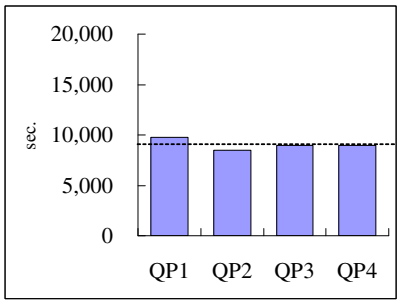

Figure 5. Query processing time on each QP with data relocation

Table 2. Total CPU times for query processing

	no relocation	with relocation
total CPU time (in sec.)	15,873	13,143

4.2 Evaluation of Data Partitioning

In order to confirm our data partitioning algorithm, we compare ours to two other partitioning algorithms: one considers only size, and the other considers only structure. In the cases of ours and the size oriented one, the original XML data is decomposed into 16 fragments, and each of four XML fragments is stored into the same QP. On the other hand, in the algorithm which considers only structure, the original XML data is decomposed into 11 fragments, and are distributed to four QPs, because it is quite difficult to divide into arbitrary number of fragments by the structure oriented algorithm. However, we tried to distribute the fragments to QPs, so that each QP has almost the same size of fragments. We then measure query processing times of each algorithm in the same way as the previous experiment.

Table 3 compares total CPU times of three data partitioning algorithms. In this experiment, 2,400 queries were also executed. This table shows that our partitioning algorithm obtains the best query processing performance of the three algorithms.

Table 3. Total CPU times for query processing

	Size and Structure	Size	Structure
total CPU time (in sec.)	13,143	13,603	19,778

4.3 Experimental Results and Discussion

From the result of the experiment shown in Section 4.1, it is clear that the dynamic data relocation improves the efficiency of query processing in distributed environments. Since query processing times of XML fragments on QPs usually differs, our system may become inefficient if queries unevenly concentrate on a specific QP. In our system, in order to achieve load balancing among QPs, QC finds fragments d_L and d_S and asks the QPs associated with d_L and d_S to exchange them each other (see Section 3.5). This relocation brought about 17% performance increase.

On the other hand, our data partitioning algorithm can obtain the best performance for query processing in the three algorithms from the experimental result. Compared with the size oriented data partitioning algorithm, there was not obvious difference from ours. However, the size oriented one may lead to additional CPU cost when data structure reconstruction is necessary. For the structure oriented partitioning algorithm, there is the possibility that larger fragments may be more frequently accessed, resulting in inefficient query processing performance due to unbalanced accesses to fragments.

From these two experimental results, it proves that we can realize efficient query processing in distributed environments if we consider both the size and structure to decompose XML data. It is, in particular, important to give consideration to the size when decomposing XML data. If we do not consider the size, the data relocation causes skew in storage cost. Though our data partitioning algorithm takes only vertical fragmentation into consideration, it is possible to apply horizontal fragmentation approaches [1, 2]. However, the size problem must be paid attention to carefully. Otherwise, both storage and CPU costs cannot be balanced when taking data relocation into account. It is worthwhile to note that the query processing using our data partition, distribution, and dynamic relocation techniques work well, even if query workload of each QP is unpredictable.

Moreover, our approach shows the nearly linear scalability at least up to 100MB of XML data according to other measurements in performance when the data size changes. Therefore, the fundamental techniques we proposed in this paper can widely be acceptable for efficient query processing for large XML data.

5 Conclusion and Future Work

In this paper, we propose methods for partitioning and distributing XML data for digital library systems. Our methods help us to execute efficient query processing for large-scale XML data using multiple computation nodes. Owing to distributed environments, our approach can obtain several advantages including parallel processing, load balancing, scalability and redundancy. Actually, our experimental results show data partitioning algorithm and dynamic relocation proposed in this paper can provide more efficient query processing than the previously known alternatives.

In future work, we plan to explore a method to automatically determine the schedule of data relocation on QPs. We believe that this extension would be able to obtain better performance than the current implementation.

Acknowledgment

This work was partly supported by CREST Program of JST, JSPS (Grant-in-Aid for Scientific Research (A) #15200010), and MEXT (Grant-in-Aid for Young Scientists (B) #17700109).

References

[1] A. Andrade, G. Ruberg, F. Baião, V. P. Braganholo, and M. Mattoso. Efficiently Processing XML Queries over Fragmented Repositories with PartiX. In *Proc. of the 2nd International Workshop on Database Technologies for Handling XML Information on the Web (DataX'06)*, pages 14–25, March 2006.

[2] J.-M. Bremer and M. Gertz. On Distributing XML Repositories. In *Proc. of the 6th International Workshop on the Web and Databases (WebDB2003)*, pages 73–78, June 2003.

[3] L. Harada and M. Kitsuregawa. Dyanamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems. In *Proc. of the 4th International Conf on Database Systems for Advanced Applications (DASFAA'95)*, pages 246–255, April 1995.

[4] K. A. Hua and C. Lee. An Adaptive Data Placement Scheme for Parallel Database Computer Systems. In *Proc. of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 493–506, August 1990.

[5] K. Kido, T. Amagasa, and H. Kitagawa. Processing XPath Queries in PC-Clusters Using XML Data Partitioning. In *Proc. of the 2nd International Special Workshop on Databases for Next-Generation Researchers (SWOD 2006)*, pages 114–119, April 2006.

[6] F. Peng and S. S. Chawathe. XSQ: A Streaming XPath Engine. *ACM Transactions on Database Systems*, 30(2):577–623, June 2005.

[7] World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`. W3C Recommendation 07 April 2004.

[8] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fourth Edition). `http://www.w3.org/TR/2004/REC-xml-20060816/`. W3C Recommendation 16 August 2006, edited in place 29 September 2006.