An Algorithm for Parallel Holistic Twig Joins on a PC Cluster

Imam MACHDI†, Toshiyuki AMAGASA†,††, and Hiroyuki KITAGAWA†,††

† Graduate School of System and Information Engineering
†† Center for Computational Sciences
University of Tsukuba
Tennodai 1–1–1, Tsukuba, Ibaraki, 305–8573 Japan
E-mail: †machdi@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

Abstract　In this paper, we propose an algorithm for parallel holistic twig joins executed on a PC cluster, especially for achieving high intra query parallelism. We deal with data redistribution in the case of workload imbalance existence in the current data allocation. The data redistribution scheme exploits containment properties of a positional representation of XML nodes to partition streams of XML nodes stored in XML databases and redistribute them to cluster nodes on the fly. In the preliminary experiment, we demonstrate the significantly improved parallel performance in terms of speed up measurement.

Key words　XML data redistribution, twig joins, intra query parallelism

1. Introduction

XML has become the de facto standard for data representation and exchange over the Internet. Nevertheless, along with the increasing size of XML documents and complexities to evaluate XML queries, existing query processing performance in a single-centralized environment will deteriorate. Parallelism is, thus, a viable solution.

One possible approach of exploiting parallelism is to adopt a PC cluster system, in which tens of commodity PCs are interconnected with a high-speed network, due to its recent popularization and commercialization. As illustrated in Figure 1, in such a system, a set of XML data being queried are distributed across cluster nodes in advance. When a query is incoming, a node acting as the coordinator analyzes it and generates a query plan consisting of several subqueries. The coordinator then forward the subqueries to respective cluster nodes. The cluster nodes compute their subqueries, and send the partial results back to the coordinator.

In the case of inter query parallelism where several different queries are executed in parallel, XML data and queries are distributed in such a way that the overall load of query processing in each cluster node will have an equal balance. As an example, Figure 1 illustrates the distribution of XML data and queries where each node has a nearly balanced load. Node 1 is responsible for executing query 1 for XML document 1, query 2 for XML document 1 and query 3 for XML document 3. Node 2 maintains processing query 3 for XML document 3 and query 4 for XML document 2, while node 3 merely processes query 5 for XML document 3. This query

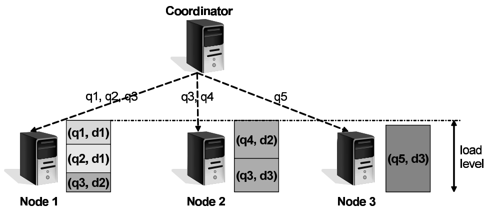


図 1　An XML query processing system.

system performs very well when all queries are executed in parallel at a time. However, a problem may arise at any time when only a single or some of the queries is requested to process. For example, only query 5 is being processed in the system, the performance of this typical query system suffers from executing a single long-running query on a cluster node while other nodes are idle.

In this paper, to overcome the above problem we focus on a single query execution for intra query parallelism with the existing pre-distributed XML data. As for the XML query processing technique, we adopt a holistic twig joins technique [2], which is an important family of XML join processing algorithms that enable us to process an XML query consisting of several branches (query twigs) holistically by scanning over XML node streams. The main objective of this research is to improve the query system performance by redistributing the existing XML data on the fly to idle cluster nodes for achieving high intra query parallelism.

Basically, our proposed redistribution method comprises two stages: a partition plan and a distribution plan. The partition plan exploits the positional properties of XML node streams that reflect the relationships between query nodes. Initially, a range of a stream, which is associated with a query node, is selected to be partitioned by a fixed window size. The computation of positional properties are propagated from the selected stream towards the adjacent connected streams by satisfying the relationships between the associated query nodes. In addition, the distribution plan allocates the partitioned streams to idle cluster nodes in the system for execution.

We make the following contributions in this paper:

- We propose a redistribution method to efficiently utilize idle cluster nodes for intra query parallelism.
- We construct an algorithm for a partition plan and a distribution plan.
- We show the effectiveness of the proposed method in terms of the parallel system performance.

The rest of the paper is organized as follows. Section 2 discusses the related work. In Section 3, we present the preliminary that underlies our work. Section 4 proposes the redistribution method. Section 5 reports experimental results. Finally, we conclude our work in Section 6.

2. Related Work

Our prior work [3] related to XML data partition for parallel holistic twig joins processing aims mainly at inter query parallelism. It organizes and maintains XML metadata, which is the basic information related to XML documents, tags and queries, in the form of data cube. The data cube represents the conceptual model for partitioning and distributing XML data. A cost model is utilized to estimate costs of query processing for XML documents and it serves as the basis for distributing queries and XML data to cluster nodes.

In the work we devise our parallel processing system comprising one cluster node selected as the coordinator and the rest as processing nodes. The global data cube describing the entire XML data and query distribution in the system is maintained by every cluster node, while every processing node also maintains its own local XML database as the result of XML data partition. When the coordinator receives an incoming query, it generates a query plan and dispatches the query or its subqueries to certain processing nodes according to the information stored in the data cube. Subsequently, the intended processing nodes receive the query, inquire the related XML data from the data cube and retrieve the related XML data from the XML database for query execution. Solutions of the query are, then, forwarded to the coordinator to be delivered to the user.

The work also states about the execution sequence of the holistic twig joins processing. Principally, the holistic twig joins algorithm operates in two phases. The first phase computes solution extensions and generates subquery (root-to-leaf paths) solutions. Solution extensions are candidate nodes that are guaranteed to give solutions to individual subqueries. The root-to-leaf paths are illustrated in Figure 2 (c). In the second phase these subquery solutions are merge-joined to compute the answers of the query twig pattern. When the query plan generated by the coordinator does not yield subqueries, the first phase and the second phase are computed by the processing nodes. Otherwise, the processing nodes compute subqueries for the first phase only and send the subquery solutions to the coordinator. In this case, the coordinator is responsible for computing the second phase for generating the query answers.

Other works [6], [7] in parallel databases deal with data skew when joining relations, even though relations are already partitioned and distributed in balanced manner. This issue is handled by basically performing redistribution, reduplication and rescheduling techniques when a query is incoming. The research has shown that redistribution plan has lead to nearly linear speed up on shared-nothing systems under even balancing condition.

3. Preliminaries

In this section, we present a brief introduction of some concepts related to holistic twig joins in [2].

3.1 XML Data Model

An XML document is a rooted, ordered, labeled tree, where each node corresponds to an element and the edges representing (direct) element-subelement relationships. Node labels consist of a set of (attribute, value) pairs, which suffices to model tags, PCDATA contents, etc. Figure 2 (a) shows the tree representation of a sample XML document.

3.2 Query Twig Patterns

A query twig pattern Q is a node-labeled tree pattern with elements and string values as node labels and its edges represent parent-child or ancestor-descendant relationships as shown in Figure 2 (b). It can be decomposed into a set of root-to-leaf path patterns as illustrated in Figure 2 (c).

3.3 Representation in XML Database

As used in [1], [2], [8], the position of every string occurrence in an XML document is represented as a 3-tuple (DocId, LeftPos, Level). Similarly, the position of every element occurrence is as a 3-tuple (DocId, LeftPos : RightPos, Level), where (i) DocId is the identifier of the document; (ii) LeftPos and RightPos can be generated by counting word numbers from the beginning of the document DocId until the start and the end of the element, respectively; and (iii) Level is

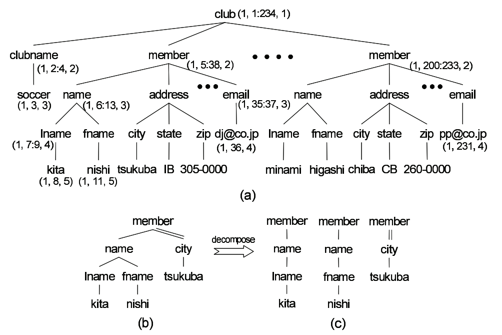


図 2 (a) An XML tree representation, (b) a query twig pattern, (c) root-to-leaf query patterns.

the nesting depth of the element or the string value in the document. By having this enumeration fashion, structural relationships of parent-child and ancestor-descendant can be determined easily and will be explained in the next subsection.

An XML database stores the entire nodes of the entire XML documents where each node is represented as a 3-tuple. Associated with each node in a query twig pattern, there is a stream, which is a sequence of nodes retrieved from the XML database with an order by $(DocId, LeftPos)$. The operations over streams are **eof** to indicate the end of a stream, **advance** to move to the next node in a stream, **next** to return the next node, **front** to return the front node of a stream, **last** to return the last node of a stream, **nextL** to return the $LeftPos$ of the next node, **nextR** to return the $RightPos$ of the next node, **nextMostL** to return the pair $mostLeft$ of $(DocId, LeftPos)$ obtained from the next node in a stream and **nextMostR** to return the pair $mostRight$ of $(DocId, RightPos)$ obtained from the next node in a stream.

3.4 Partitions

A partition of a stream is defined as a substream whose node sequence follows the same order by $(DocId, LeftPos)$. Partitions of a stream are also sorted by their (*mostLeft*, *mostRight*) positions.

3.5 Positional Properties

By having positional representation, we extend properties as defined in [8] as follows:

Containment An occurrence of an ancestor node a and a descendant node d satisfies: $a.DocId = d.DocId$, $a.LeftPos < d.LeftPos$, and $d.RightPos < a.RightPos$.

Direct Containment An occurrence of a parent node p and a child node c satisfies: $p.DocId = c.DocId$, $p.LeftPos < c.LeftPos$, $c.RightPos < p.RightPos$ and $p.Level + 1 = c.Level$.

Left Containment An occurrence of an ancestor node a and a descendant node d satisfies: $a.DocId = d.DocId$ and $a.LeftPos < d.LeftPos$.

Right Containment An occurrence of an ancestor node a and a descendant node d satisfies: $a.DocId = d.DocId$ and $d.RightPos < a.RightPos$.

Note that the containment property describes the ancestor-descendant structural relationship, while the direct containment property describes the parent-child relationship. Also, the direct containment property is a special case of the containment property. We can see in Figure 2 that the position (1, 7:9, 4) of a descendant node *lname* is contained in the position (1, 5:38, 2) of an ancestor node *member*. The position (1, 7:9, 4) of a child node *lname* is directly contained in the position (1, 6:13, 3) of its parent node *name*.

3.6 System Environment

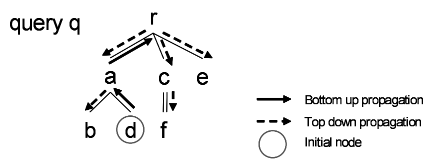
As described in Section 2, we utilize the same parallel processing system in our prior work [3] with additional functionalities. The coordinator sends running query states along with an incoming query to the intended processing nodes which, then, check the states against the data cube to identify idle cluster nodes. Additional communications among cluster nodes are required to dispatch partitioned streams from a processing node to other idle processing nodes. At last, the coordinator collects solutions from processing nodes and performs merge-join operations to produce the final solutions.

4. The Proposed Redistribution Method

In this section, we present our proposed redistribution method starting with the observation of a twig tree to construct partitioned streams. The aim is to partition streams on the fly where each partition represents XML twig trees to be matched with the query twig patterns. We strive for minimizing the time required to compute the partitions and having no communication dependency of partitioned streams among cluster nodes while processing the holistic twig joins.

Considering XML node streams that are associated with nodes of a query twig pattern, they implicitly represent XML twig trees that are associated with the query twig pattern. We observe that generally the root node of the query has the smaller stream size and its descendant nodes have larger stream sizes. It is easier and more desirable to start partitioning streams from the query root node as the initial node, then propagate the partitions to its children nodes until leaves nodes, but only if the stream size of the root node is large enough to be partitioned. To avoid this case, however, we may select an initial node having the largest stream size to be partitioned first, then propagate the partition to the root node and leaves nodes.

Regarding nodes distribution, we also notice that positions



```
PartitionPlan()
n ← getNodeWithMaxStreamSize(q)
Partitions[n] ← EqualPartitioning(Streams[n], wdwsize)
//compute partitions from node to root of query
while n ≠ root(q) do
    p = parent(node)
    Partitions[p] ← BottomUp(Partitions[n], Streams[p])
    n ← p
end while
//compute partitions from root to all nodes of query
Visit(root(q), root(q))
```

```
Visit(q, p)
if Not isRoot(q) AND Partitions[q] ≠ NULL then
    if isDuplicate(Streams[p]) then
        Partitions[q] ← EqualPartitioning(Streams[q], wdwsize)
    else
        Partitions[q] ← TopDown(Partitions[p], Streams[q])
end if
for all child in children(q) do
    Visit(child, q)
end for
```

図 3 Partition propagation and algorithm of PartitionPlan.

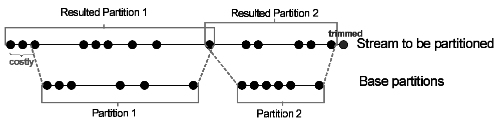
of nodes in a stream are not always uniformly distributed within the stream range (*mostLeft* to *mostRight*). Instead of partitioning streams based on an equal range, the selected initial stream is partitioned according to an equal number of nodes constituted as the initial window size. If the initial window size is too large, the distribution of partitioned streams may possibly lead to data skew. On the other hand, if the window size is too small, it requires more time to compute partitions. In this study, our initial attempt is to select several candidates of initial window sizes and adopt a distribution method to avoid the problem of data skew.

4.1 Partition Plan

By given a query twig pattern along with their streams, a selected initial node and an initial window size, the main task is to compute partitions of streams started from the initial node propagated to the rest of nodes in the query twig pattern such that the distribution of partitions to cluster nodes does not introduce data dependency when computing parallel holistic twig joins.

Based on partitions of the initial stream, other streams associated with other query nodes can be eventually partitioned by means of propagation. As shown in Figure 3, firstly we propagate partitions from the initial node to the root node of the query by applying a bottom-up partition approach. Subsequently, stream partitions of the root node propagate partitions to all other unvisited nodes by applying a top-down partition approach.

In the Bottom-Up partition approach a parent's (ancestor's) stream is partitioned according to the containment property of the base stream partitions. As an example of bottom up partition illustrated in Figure 4, although it is desirable to find all nodes in the intended stream to contain nodes in each base partition as indicated by dashed lines, we deliberately limit the searching effort since it is the most influential cost in this stage. In this case, the *mostLeft* of the intended partition is obtained from either the current or the next node pointed in the stream. We just try to find the *mostRight* which is the first found node that satisfies the right containment property or the last found node that satisfies the containment property. If the range of the intended stream to be partitioned is much longer than of the base partition, it is possible that some nodes at the end of the stream may not be included in the partitions. This contributes to more efficient processing of the holistic twig joins. Finally, stream nodes whose positions are within the range of *mostLeft* and *mostRight* are copied to a partition.



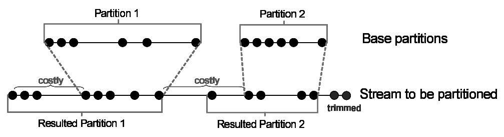
```
BottomUp (basepartitions, stream)
for all part in basepartitions do
    if (Not isLeftContainment(front(part), next(stream))) then
        advance(stream)
    end if
    mostL ← nextMostL(stream)
    SearchFirstRightContainment(last(part), stream)
    SearchLastContainment(last(part), stream)
    mostR ← nextMostR(stream)
    CopyStreams(ancpart, stream, mostL, mostR)
    ancpartitions ← ancpartitions U ancpart
end for
return ancpartitions
```

図 4 An example and an algorithm of Bottom Up.

The Top-Down partition approach has similar mechanism with the Bottom-Up approach to partition the intended stream by finding the containment property. For each partition, the *mostLeft* is obtained from the current node or the next node pointed in the intended stream. The *mostRight* position is obtained from searching the last node in the intended stream that satisfies the right containment property. This approach also has an advantage to trim some left-over stream nodes that are not included in the partitions. Figure 5 describes an example of top down partition approach and the algorithm.

4.2 Distribution Plan

The main task of this stage is to distribute partitions to idle cluster nodes and balance the workloads by utilizing Round



```
TopDown (basepartitions, stream)
for all part in basepartitions do
    if (Not isLeftContainment(front(part), next(stream))) then
        advance(streams)
    end if
    mostL ← nextMostL(stream)
    SearchLastRightContainment(last(part), stream)
    mostR ← nextMostR(stream)
    CopyStreams(descpart, stream, mostL, mostR)
    descpartitions ← descpartitions U descpart
end for
return descpartitions
```

図 5 An example and an algorithm of Top Down.

Robin approach. As the time is very crucial for "on the fly" distribution, in the case of imbalance workloads after distribution we do not make further attempts to rebalance the workloads.

Initially this stage estimates the amount of workloads to be allocated to other cluster nodes. The running query states dispatched by the coordinator are analyzed against the data cube to identify every cluster node that is currently running queries and to estimate the fraction of its busy time bs, which means the idle time can be computed as (1 - bs). The workload of a cluster node is measured in terms of the number of partitions to be allocated with the proportion of its idle time over the entire idle time of all cluster nodes N. Hence, the workload of a cluster node P_i is simply estimated with the following equation:

$$WL(P_i) = \frac{1 - bs_i}{\sum_{i=1}^{N} 1 - bs_i} \#partitions \qquad (1)$$

Round Robin distribution is conducted to allocate partitions to every cluster node according to its estimated workload. Finally, allocated partitions are sent to their respective cluster node destinations.

5. Experimental Evaluation

The main objective of this preliminary experiment is to show the contribution of data redistribution towards the parallel speed up performance.

5.1 Experiment Platform and XML data

The experiment platform used is a shared-nothing homogeneous cluster system. One node plays a role as the coordinator and 9 nodes as the processing nodes. Each node has a 4-ways Intel Xeon(TM) 3.0 GHz CPU with 1 GB memory running RedHat Enterprise Linux 4.0. PostgreSQL 8.1 is installed as the XML database in each node. All nodes are connected through a Gigabit high-speed LAN and we use MPICH2 for implementing the communication among cluster nodes.

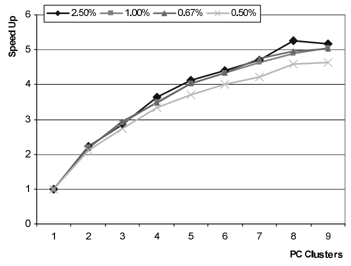


図 6 Speed up performance of query 1.

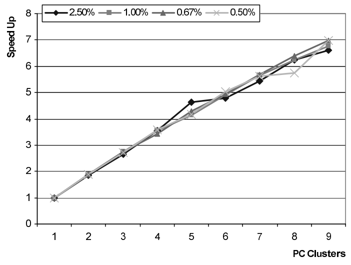


図 7 Speed up performance of query 2.

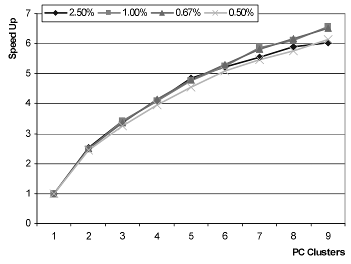


図 8 Speed up performance of query 3.

We adopt XML Benchmark Project [5] proposed by German CWI as our experiment data set. The XMark data set provides a large single scalable document. The test data size is 110M of a single XML document and we provide three query twig patterns with different structure complexities: simple, medium, and complex as shwon in Appendix. The total stream sizes required to execute query 1, query 2, and query 3 are 99,250 nodes, 50,917 nodes and 181,268 nodes respectively.

5.2 Experimental Results and Evaluation

In the experiment, we simply set one cluster node to store the XML data in an XML database and to redistribute to other cluster nodes. We set four relative window sizes (2.5%, 1%, 0.67%, 0.5%) over a stream size to determine the window sizes. The larger relative window size indicates the larger window size. The measurement of the speed up performance, or equivalently the running time, gives an immediate measure of the effectiveness of our proposed redistribution method.

We can see in Figure 6, 7, and 8 that the best speed up performances of the three queries have different relative window sizes. The relative window size with 2.5% value gives the best speed up performance for the execution of query 1, while the best speed up performance of query 2 is contributed by the 0.67% relative window size. For the execution of query 3, the relative window sizes of 1% and 0.67% contribute relatively the same speed up performance. In this case, window size plays an important role of contributing workload balance that gives impact on the performance.

In addition, the execution of query 1 gains the smallest speed up performance, while the execution of query 2 gains the highest. On average, the computation time of data redistribution for query 1 contributes 27% of the total execution time. Meanwhile, the computation time of data redistribution for query 2 and 3 is about 18% of the total execution time.

Although we suspect there may be inefficiency in implementing our proposed redistribution method, in overall the data redistribution method contributes to significant impact on good parallel speed up performance.

6. Conclusions and Future Work

In this paper, we proposed an "on the fly" redistribution method comprising two stages: partition plan and distribution plan. In the partition plan stage, positional properties are exploited to compute partitions by utilizing bottom-up and top-down partition approaches. In the distribution stage, partitions are distributed to cluster nodes by estimating workloads for better balance. Our experimental results showed good speed up performance.

As for future work, we plan to further investigate the most appropriate window size by considering the query structure, data distribution, and system capacity. Also we will exploit different strategies to resolve inefficiency in the implementation.

Acknowledgments

This study has been partially supported by MEXT (#19024006), Grant-in-Aid for Young Scientists (B) (#19700083) and CREST of JST (Japan Science and Technology Agency).

文　　献

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 141–, 2002.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[3] I. Machdi, T. Amagasa, and H. Kitagawa. Cube-based analysis for maintaining xml data partition for holistic twig joins. *Journal of the Database Society of Japan*, 7(1):121–126, June 2008.

[4] R. Sakellariou and J. R. Gurd. Compile-time minimisation of load imbalance in loop nests. In *Proceedings of the 11th International Conference on Supercomputing*, pages 277–284, 1997.

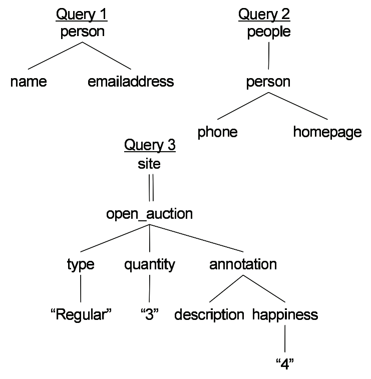
[5] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: a benchmark for xml data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

[6] J. L. Wolf, D. M. Dias, and P. S. Yu. A parallel sort merge join algorithm for managing data skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):70–86, 1993.

[7] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1043–1052, New York, NY, USA, 2008. ACM.

[8] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of data*, pages 425–436, 2001.

Appendix


Query 1
person
name
emailaddress
Query 2
people
person
phone
homepage
Query 3
site
open_auction
type
quantity
annotation
"Regular"
"3"
description
happiness
"4"