Querying XML Data by Parallel Holistic Twig Joins Processing

Imam MACHDI[†], ***Student***, **Toshiyuki AMAGASA**[†], ***and*** **Hiroyuki KITAGAWA**[†], ***Advisors***

SUMMARY In this study we develop a parallel XML query processing system on a shared-nothing cluster system to process query twig patterns on XML data based on a holistic twig joins algorithm. The study focuses on finding new novel schemes of XML data partition, which is one of the main technical aspects for good system performance in the parallel processing context, for both static and dynamic XML data distribution. For static XML data distribution, we propose a partitioning scheme called Grid Metadata Model for XML (GMX) that provides an abstraction of performing XML data partition. This partitioning scheme outlines a set of partition methods that provide different levels of partition granularities such as document cluster, query cluster, document, query and query-path partitioning methods. During static XML data distribution onto cluster nodes, different partition granularities from the coarsest to the finest partitions are utilized to balance workloads among the cluster nodes. In addition, for dynamic XML data distribution, we propose a streams-based partitioning method that provides much finer partition granularity than the former methods. When queries being executed in the system introduces workload imbalance, the method is executed on-the-fly.

Keywords: *XML data partition, parallel holistic twig joins*

1. Introduction

With the growing popularity of the World Wide Web, many applications have adopted XML as a de facto standard format for representing various kinds of information. Greater volume of XML data, increasing number of concurrent users and more complex queries are challenges toward the performance capability of such a query processing system.

Many research activities have devised XML query processing algorithms to deal with these challenges. Among other XML query processing algorithms introduced by [1], [7], [17], and [18], the family of holistic twig joins algorithms introduced in [4], [5], and [10] has distinguished features. The underlying structure of the holistic twig joins is typically in the form of streams, which are sequences of XML nodes represented in a 3-tuple representation [18] that is able to specify structural relationships among stream nodes. In addition, the holistic twig joins have capabilities of performing multiple scans over stream inputs simultaneously, reducing redundant query root-to-leaf path solutions optimally and skipping stream nodes that do not contribute solutions. These distinguished features lead to better query processing performance [6]. Due to this performance advantage, they are notoriously regarded as important XML query processing algorithms.

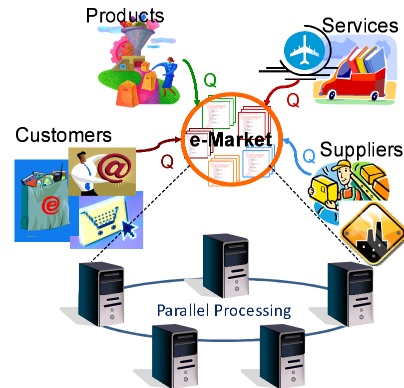

Fig. 1 E-Market application.

While a single processing PC suffers from limitations of storage space, main-memory and processing power that prevent from processing such large XML data, parallel query processing continues to be important to give timely response. In a shared-nothing PC cluster system, multiple PCs communicate via a high-speed interconnect network and each PC has its own private memory and disk(s). XML data to be queried are partitioned and distributed across cluster PCs in advance. When a query is incoming, one cluster PC acting as a coordinator analyzes it and generates a query plan consisting of several sub-queries. The coordinator, then, forward the sub-queries to respective cluster PCs for processing. The results of processing are, then, returned back to the coordinator.

As an illustrative example in Figure 1, an e-Market application typically manages heterogeneous XML documents and numerous queries. The XML documents vary in their sizes from several KBs to hundreds of MBs. Also, they describe various information related to products, services and business entities such as customers, distributors, suppliers and partners. Users from different business entities submit queries to the application system to inquire specific information for their own interests. The submitted queries may have different complexities and different processing time. To optimize the query processing performance on a parallel cluster system, the entire XML data needs to be partitioned and distributed onto cluster nodes such that the overall workloads are about balanced among the cluster nodes.

[†]Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba

In fact, due to the nature of semi-structured XML data, traditional partitioning techniques cannot serve well for XML data; therefore, it becomes a major technical challenge. Many studies have proposed several techniques of XML data partition mainly based on XML structure. Our prior works [2], [8] have proposed schema graph decomposition by utilizing XPath approach for mapping XML data to relational tables. Similarly, Bremer and Gertz [3] use a tree-structured schema to represent an XML document and to extract its rooted node paths. The paths are, then, indexed and stored in a RepositoryGuide. WIN [16] proposes a partitioning technique based on XML sub-trees where upper nodes of the tree are duplicated to all processing nodes and the remaining sub-trees are partitioned to different processing nodes. Similar to WIN the work of Kurita et al. [9] introduces their XML data partitioning technique based on XML subtree structure by specifying a permissible size range for subtree-fragment sizes.

In this study we propose new XML partitioning schemes for parallel holistic twig joins processing in the context of large-scale XML repositories that manage heterogeneous XML documents and numerous queries. The main objective of our partitioning scheme is to partition and distribute streams of XML nodes through an abstraction of the grid metadata model for XML (GMX) and streams-based partition model for static data distribution and dynamic data distribution, respectively. GMX describes relationships between streams of XML nodes and queries, facilitates a partitioning scheme comprising XML document clustering, query clustering and partition refinement. To measure workload balance in the system, we adopt a cost model to calculate query processing costs associated with each partition. The results of distributing GMX partitions lead to inter-query parallelism as well as intra-query parallelism at some extents. Although XML data partitions are distributed statically in advance for the overall workload balance, executing some queries may lead to workload imbalance indicated by less processing time or even idle on certain cluster nodes. To overcome this situation, the streams-based partition method is utilized for XML data redistribution. This method leads to improving intra-query parallelism.

2. Preliminary

In this section, we present a brief introduction of some concepts related to holistic twig joins in [4].

2.1 XML Data Model and XML Database

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. Every node occurrence in an XML document is labeled with a 3- tuple representation. The position of every string occurrence is represented as $(DocId, LeftPos, Level)$. Similarly, the position of every element occurrence is as $(DocId$, $LeftPos$: $RightPos$, $Level)$, 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 until the start and the end of the node, respectively; and (iii) $Level$ is the nesting depth of a node in the XML document. By having this node representation, structural relationships between nodes describing ancestor-descendant and parent-child relationships can be determined easily.

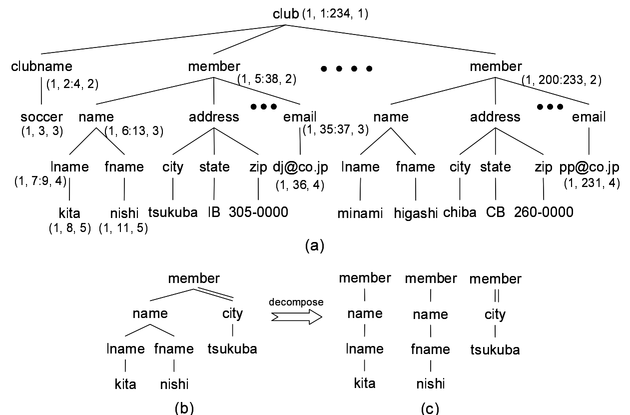

Fig. 2 (a) XML tree representation, (b) a query twig pattern and (c) query root-to-leaf paths.

An XML database stores all 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 and ordered by $(DocId$, $LeftPos)$.

2.2 Query Twig Patterns

A query twig pattern or a query for short 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). In this work, a query has statistics that includes α as a probability of query occurrences, β as estimated root-to-leaf output sizes, and γ as estimated final output sizes. We assume that query statistics can be obtained by monitoring the system.

A query can be decomposed into a set of root-to-leaf path patterns or query paths. Each query path is a simple query twig pattern and it inherits query statistics from its associated query. Illustration of query

paths is shown in Figure 2 (c).

2.3 Twig Stack Algorithm

In the holistic twig joins [4], given a query twig pattern the twig stack algorithm basically operates in two phases. In the first phase, firstly, it computes solution extensions, which generate candidate nodes that are guaranteed to give solutions to individual query root-to-leaf paths. It traverses a set of input streams of nodes, which correspond with the query nodes, to match their structure relationships at any position in streams with the query pattern. Secondly, the algorithm generates solutions to individual query root-to-leaf paths. It constructs stack encodings that resemble individual query root-to-leaf path patterns for storing only candidate nodes generated by the solution extensions. It guarantees that the solutions are generated in the order of common prefixes ($DocId$, $LeftPos$) of query root nodes.

In the second phase, these solutions are merge-joined to compute the answers to the query twig pattern. The algorithm merges query root-to-leaf path solutions whose query roots have common prefixes ($DocId$, $LeftPos$). All merged query root-to-leaf path solutions that satisfy the query twig patterns are generated for the answer of the query twig pattern.

2.4 Parallel Query Processing Model

For maintaining heterogeneous XML documents and processing numerous query twig patterns, we devise the query processing model mainly for inter query parallelism. The structure of our cluster system comprises one coordinator node and a number of processing nodes. They maintain the global XML metadata including the global distribution information for determining query processing plans. Processing nodes maintain their own XML databases resulted from partitioning XML data. Among processing nodes there is no data dependency that requires communication. The communication occurs only between the coordinator and processing nodes when the coordinator sends a query to processing nodes and receives solutions from them.

There are two particular cases in processing queries: (i) A whole query is processed by one or more processing nodes. The processing nodes compute all tasks in the first phase and the second phase of the twig stack algorithm and deliver the final solutions to the coordinator. (ii) A query is decomposed to its root-to-leaf paths. Each path is processed by one or more processing nodes. The processing nodes compute only tasks in the first phase of the twig stack algorithm and deliver solutions of individual root-to-leaf paths to the coordinator. Finally, the coordinator merges the root-to-leaf path solutions for the query answers.

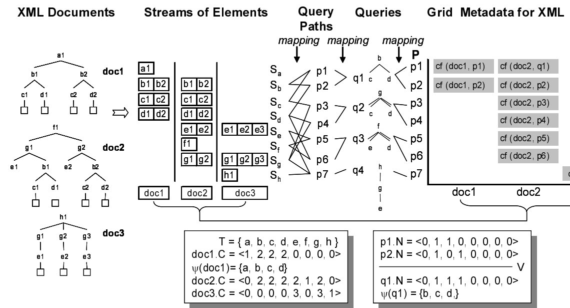

Fig. 3 An overview of constructing the grid metadata for XML.

3. Grid Metadata Model for XML

In this section, we will explain Grid Metadata Model for XML (GMX) [13], [12] and its series of XML data partitioning methods. The main objective is to partition streams of XML nodes, which are the underlying data structure for holistic twig joins, and to provide ways of refining partitions for achieving workload balance in the distribution.

3.1 Constructing GMX

GMX is a model for XML metadata that is maintained in a two-dimensional structure for describing a relationship between XML documents represented as XML node streams and twig queries. The main objective of this model is to provide abstraction for partitioning streams by taking into account the coherency between query twigs and XML documents. In this subsection, all examples used refer to Figure 3.

Figure 3 illustrates the relationship among XML documents, streams of elements and query twig patterns. As explained in the previous section, an XML database, which stores nodes of XML documents in the form of 3-tuple representation, generates all distinct streams of elements, e.g., S_a, S_b, S_c, S_d, S_e, S_f, S_g, and S_h. In this case, S_a is associated with doc_1 only, S_b is associated with doc_1 and doc_2, while S_e is associated with doc_2 and doc_3. On the other hand, query root-to-leaf paths (query-paths for short), which are decomposed from their related query twig patterns, have their own elements that can be associated with streams of elements. For instance, path p_1 that has elements b, c is associated with S_b and S_c and, similarly, p_2 that has elements b, d is associated with stream S_b and S_d. Transitively, a query twig pattern have a relationship with streams of elements through the relationship between query paths and streams of elements. For instance, query q_1 is associated with S_b, S_c, and S_d because its root-to-leaf paths p_1 and p_2 are associated with S_b, S_c, and S_d. Moreover, another transitive relationship occurs between query twig patterns and XML

documents, e.g., query q_1 is obviously associated with doc_1 and doc_2 because S_b, S_c, and S_d are associated with doc_1 and doc_2.

By observing this relationship illustration, the partitioning model has a document dimension D, a query-path dimension P, and a query dimension Q, where elements of the query-path dimension P can be aggregated for their associated query in the query dimension Q. The association of an instance in the query-path dimension and an instance in the document dimension indicates a processing cost of the query-path for the XML document; the cost function is specified in Eq. 3. Similarly, the association of an instance in the query dimension and an instance in the document dimension indicates a processing cost of the query for the XML document; the cost function is specified in Eq. 2.

We define a set of m-distinct element names, which can be simply derived from the entire XML documents, e.g., $T = \{a, b, c, d, e, f, g, h\}$. Each element in T is stored in no particular order. However, once T is constructed its element sequence is fixed because element positions in T will be referred by the dimensions.

The document dimension D is defined as a set of documents, e.g., $D = \{doc_1, doc_2, doc_3\}$. Each document doc_i has a m-dimensional vector C containing the number of occurrences of every element name t_j in the document doc_i. Also, there exists a function $\psi(doc_i)$ to return a set of element names in T that correspond to non-zero component values of the vector C.

The query-path dimension P is defined as a set of query-paths, e.g., $P = \{p_1, ..., p_7\}$. Each query-path p_i is represented as (q, N) where q is a query twig pattern to which the query-path belongs, N is a m-dimensional vector containing the value of 1 for an element occurrence and, otherwise, 0 for no occurrence. Similarly, a function $\psi(p_i)$ exists to return a set of elements in T that correspond to non-zero component values of the vector N.

The query dimension Q is defined as a set of query twig patterns, e.g., $Q = \{q_1,...,q_4\}$. Each query q_i is represented as $(\alpha, \beta, \gamma, N)$ where α is a probability of query occurrences in the system, β is an estimated root-to-leaf output size, γ is an estimated final output size and N is a m-dimensional vector that can be constructed from its query-path vectors. In this case, $q_i.N = \bigvee_{p_j.q=q_i} p_j.N$ where $\bigvee$ denotes component-wise OR operation. Similarly, a function $\psi(q_i)$ exists to return a set of elements in T that correspond to non-zero component values of the query vector N.

Once all dimensions are completely determined, we represent GMX as $\{(d_i, p_j, pcost) \mid M : d_i \times p_j \rightarrow pcost \wedge \psi(p_j) \subseteq \psi(d_i)\}$. A function M specifies a relationship between a document d_i and a query-path p_j that functionally determine a cost $pcost$ of processing the query-path p_j for the document d_i. The relationship specification must satisfy a full containment property: *all elements of a query-path p_j denoted as $\psi(p_j)$ are fully contained in elements of a document d_i denoted as $\psi(d_i)$.* The cost $pcost$ is computed by a cost function $cf(d_i, p_j)$.

In the case that the query-path dimension is aggregated for the query dimension, the GMX is represented as $\{(d_i, q_j, qcost)\}$ where similarly, the relationship is specified as $\psi(q_j) \subseteq \psi(d_i)$ and a cost $qcost$ of processing the query q_j for the document d_i is computed by a cost function $cf(d_i, q_j)$.

3.2 Cost Model

The proposed cost model is to estimate a query processing cost for an XML document. This cost estimation is crucial part since the outcome of cost estimation will directly affect the distribution of XML data and queries. We will introduce the cost model for each query processing case mentioned in the previous subsection.

$$cf = \alpha(\frac{InputSize}{R_{I/O}} + \frac{C_{comp}}{R_{comp}} + \frac{C_{comm}}{R_{comm}}) \quad (1)$$

$$cf(d, q) = \alpha(\frac{1}{R_{I/O}} + \frac{1+3\beta+\gamma}{R_{comp}} + \frac{\gamma}{R_{comm}}) \sum_{t \in q.nodes} |S_t| \quad (2)$$

$$cf(d, p) = \alpha(\frac{1}{R_{I/O}} + \frac{1+2\beta}{R_{comp}} + \frac{\beta}{R_{comm}}) \sum_{t \in p.nodes} |S_t| \quad (3)$$

Basically, we define the cost model of a query as expressed in Eq. (1) that includes three terms of costs and α, a probability of the query occurrence. The first term is the cost of disk I/O access at start up to retrieve from an XML database the $InputSize$ which is the number of input stream nodes associated with the query nodes denoted as $\sum |S_t|$ and $R_{I/O}$ is the retrieval access rate. The second term is the computation cost C_{comp} for manipulating the retrieved nodes in memory and R_{comp} is the computation rate. The last term is the communication cost C_{comm}, especially for delivering solutions to the coordinator and R_{comm} is the communication rate in a cluster network.

We need to develop a careful investigation according to the particular cases in query processing as discussed above. The first case is to process a whole query within a single cluster node. In the first phase of the holistic twig joins, getting solution extensions requires computation linear to the $InputSize$. The computation of manipulating stacks and generating individual root-to-leaf path solutions is linear to the number of candidate nodes involved in the solutions. It can be estimated by a fraction β of the $InputSize$. Thus, the entire computation in the first phase can be simply expressed as $(1 + 2\beta)InputSize$. In the second phase, the merge-join task is linear in the sum of its input (the solutions to individual root-to-leaf paths) and output (the answer to the query); a fraction γ of the $InputSize$ is

used to estimate the number of nodes in the output. It can be simply expressed as $(\beta + \gamma)InputSize$. In addition, communicating the query answer to the coordinator obviously requires the number of nodes involved in the final answer, $\gamma InputSize$. Therefore, the whole expression of the first query case is simply stated in Eq. (2).

The second case is to decompose the query into its root-to-leaf paths p. The cost is computed only for the tasks in the first phase of the holistic twig joins. The computation of the first phase requires similar computation of the first phase in the first case. Meanwhile, the communication to deliver the solutions takes the number of nodes involved in the root-to-leaf path solutions, $\beta InputSize$. Therefore, the whole expression of the second query case can be simply stated in Eq. (3).

3.3 Partitioning XML Data

A partition is defined as a subgrid metadata and a refined partition is a subgrid metadata of a partition. For generality, a refined partition is also called a partition due to possibly further refinement. In addition, a partition is associated with a cost, which is computed by the cost function.

In the proposed partitioning scheme, firstly we attempt to group similar XML documents and similar queries in partitions to be expectedly allocated in the same cluster nodes for the purpose of query execution efficiency. For some coarse partitions, we refine them further by our partition refinement methods so that our distribution approach is able to average workloads of partitions in cluster nodes.

3.3.1 Clustering XML Documents

Clustering XML documents, the first method of XML data partitioning is to manage heterogeneous XML documents by grouping them according to their similar elements. A hierarchical clustering technique is adopted from other clustering techniques such as K-means and density-based clustering techniques because the clustering technique provides flexibility for users to decide the best clustering results easily.

For clustering XML documents, feature vectors are outlined from every document vector $doc_i.C$ in the document dimension D, where each vector value of each document is normalized towards the entire number of element occurrences. Proximities to define document similarity in the hierarchical clustering are measured by Euclidean distance with three different methods: single link, complete link and group average. Finally, each clustering result is regarded as a partition.

Figure 4 shows partitions as the result of clustering XML documents. In this figure, $doc1$ and $doc2$ are more similar in terms of their common elements than the others; so, that they reside in the same partition.

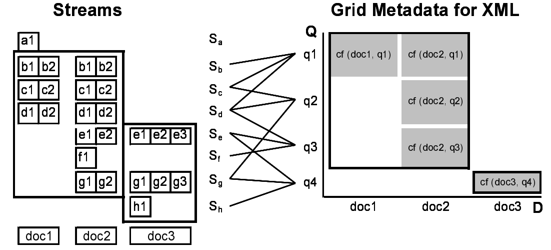

Fig. 4 Partitions resulted from clustering XML documents.

Also, the relationship between partitions in the grid metadata and streams is obviously seen.

3.3.2 Clustering Queries

In the query clustering method, we use the same hierarchical clustering technique to group queries that have similar query elements without considering the query structure. Every partition resulted from XML document clustering is further clustered by this method. The feature vectors are derived from outlining each query vector of query element occurrences $q_i.N$ without normalizing its values and Manhattan distance is more appropriate to measure the proximity. Finally, each cluster result is regarded as a refined partition.

Figure 5 shows partitions as the result of clustering queries. In this figure, q_1 and q_2 residing in doc_1 and doc_2 are more similar in terms of their common elements than q_3. So, the former partition is split horizontally into two partitions: one partition is to group q_1 and q_2 and another one is for q_3. Also, it is noticeable that the stream S_d is duplicated in the two partitions as required by q_1, q_2, and q_3.

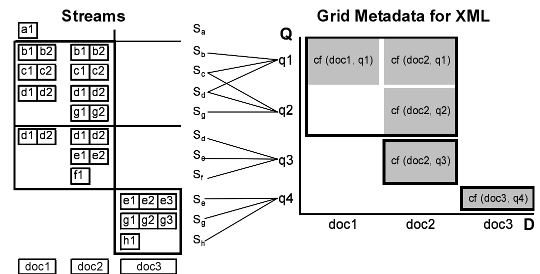

Fig. 5 Partitions resulted from clustering queries.

3.3.3 Refining Partitions

We devise further partition refinement methods. A partition having considerably high cost of processing queries may be further refined as necessary to cope with imbalance workloads. There are three methods of partition refinement. First, a partition is composed from one or more queries associated with many XML documents. The partition is split on XML documents.

Second, a partition is composed from many queries associated with one XML document. The partition is split on queries. Third, a partition is composed from one query associated with one XML document. The partition is split according to root-to-leaf paths of the query. Figure 6 illustrates results of partition refinement methods.

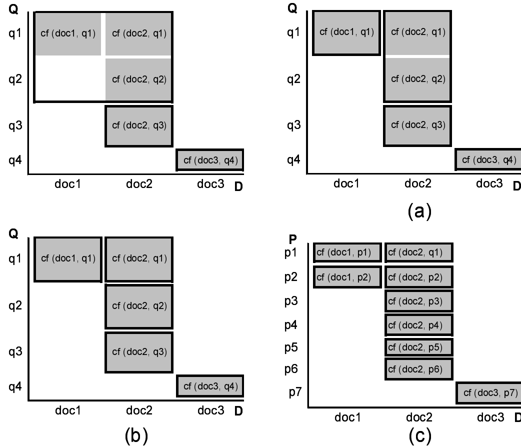

Fig. 6 Partition refinement (a) split by documents, (b) split by queries, (c) split by query-paths.

3.4 Distributing Partitions Statically

The GMX partitioning schemes are computed while XML data distribution is being performed statically before executing queries. The main objective of our distribution approach is to average workloads among cluster nodes as well as to minimize the overall query processing costs and storage costs. A workload of a cluster node is contributed from accumulated costs of partitions allocated on the cluster node. The distribution approach just gives sub-optimal workload balance since, in fact, equal workloads in all nodes are very difficult to achieve and impractical to find the best distribution strategy.

Initially, we specify n cluster nodes where each cluster node maintains its workload and a set of partitions. We specify a threshold τ, which is defined as the average workload in n cluster nodes. A cluster node is said to be overloaded if its workload exceeds the threshold value.

Basically, our distribution approach adopts a Round Robin method combined with heuristic rules with the objective of minimizing the workload variance in all cluster nodes. Round Robin method is utilized at initial distribution of partitions that are resulted from performing document clustering and query clustering. Then, repeatedly some selected partitions in overloaded cluster nodes are refined. The refined partitions are subsequently redistributed to underloaded nodes until the workload variance is minimized.

To simplify search space for redistributing and refining partitions, we implement two heuristic rules. The first rule is to redistribute partitions. Some partitions in a cluster node having the highest workload are attempted to move to a cluster node having the lowest workload only if the movement leads to a lower workload variance value. This is conducted repeatedly to the second highest workload node and so on until there is no possible movement of partitions among cluster nodes. In this rule, we try to avoid refining partitions to exploit the similarity of XML documents and queries in partitions for query execution efficiency. The second rule is to refine partitions. A partition with the highest cost in a highly overloaded cluster node is chosen for partition refinement. The refined partitions are, then, assigned to the intended underloaded cluster node only if the assignment yields a lower workload variance value. These two heuristic rules are repeatedly executed until the variance value is minimized under a certain threshold value ϵ.

4. Streams-based Partitioning Model

We propose an additional XML data partitioning method, which is based on range containment concept, to provide more refined partitions at stream nodes-based granularity. Streams-based partition introduced in [14], [11] is computed on-the-fly when incoming queries introduce imbalance workloads of query processing costs on the system. In fact, dynamic partition and distribution contributes additional costs of computing the partitions and distributing the partitions. The requirement to compute the partitions and to distribute them should be simple and straightforward to keep computation time minimal.

4.1 Basic Notion

Considering a given query twig pattern, we try to find candidate solutions on XML data by matching the query twig pattern against the structure of the XML data. In this case, candidate solutions are obtained from streams of XML nodes associated with the query twig pattern. The main intention is to compute partitions containing candidate solutions of stream nodes such that data dependency among partitions does not exist. This basic notion of computing partitions is illustrated in Figure 7.

Partition of a stream is basically a substream. It has a range positional numbers ($MostLeft$, $MostRight$) that can be obtained from the pair ($DocId$, $LeftPos$) of the first node and pair ($DocId$, $RightPos$) of the last node in the stream, respectively. We define the range containment property of two different streams S_1 and S_2 as follows: S_2 is contained in S_1 if $S_1.MostLeft < S_2.MostLeft$ and $S_2.MostRight$

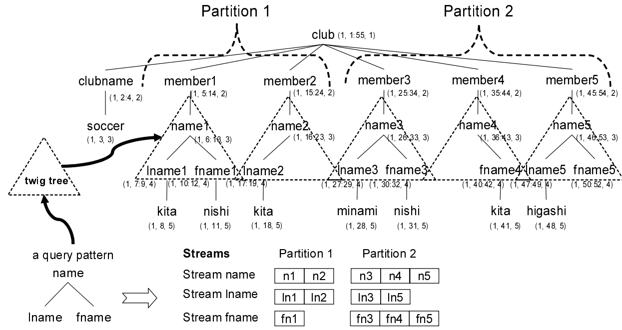

Fig. 7 The basic notions of partitioning.

$< S_1.MostRight$ where

- $S_1.MostLeft < S_2.MostLeft$ is defined if $S_1.DocId = S_2.DocId$ and $S_1.LeftPos < S_2.LeftPos$, or $S_1.DocId < S_2.DocId$,
- $S_2.MostRight < S_1.MostRight$ is defined if $S_1.DocId = S_2.DocId$ and $S_2.RightPost < S_1.RightPost$, or $S_2.DocId < S_1.DocId$.

4.2 Partitioning Streams of XML Nodes

The way to generate partitions is to select an initial stream associated with an element in a query twig pattern to be partitioned first. We select a stream having the largest size in the given query as the initial stream. We notice that positions of XML nodes in the initial stream are not always uniformly distributed within a certain range of positions. Instead of partitioning the initial stream according to an equal range of positions, the initial stream is partitioned according to a window size, which is the basic unit size of a partition containing fixed number of XML nodes.

Based on partitions of the initial stream, other streams associated with other query nodes can be eventually partitioned by propagation. Propagating partitions of a stream to another stream takes into account the structural relationship between the two query nodes associated with the streams by computing their range containment properties of the stream nodes. As shown in Figure 8, we first 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 other streams of all other unvisited nodes by applying a top-down partition approach.

In the bottom-up partition approach, a parent's (ancestor's) stream will be partitioned according to the containment property of the base stream partitions (descendants or child's stream). Here, for each partition in the base stream, we set a range $(MostLeft, MostRight)$ positions, then search the $(MostLeft, MostRight)$ positions of the ancestor's stream by computing the range containment between

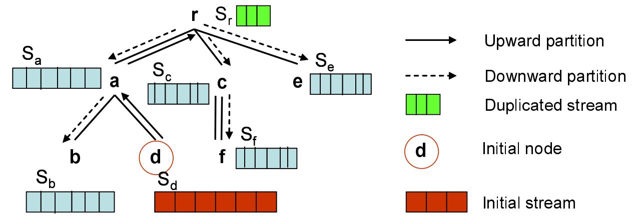

Fig. 8 Partition propagation.

partitions of the two streams.

In Figure 9 (a), the results of bottom-up partitions can be analyzed as follows: Searching the $(MostLeft, MostRight)$ positions of the ancestor's stream may contribute to some stream nodes not giving candidate solutions. This condition, however, will be handled by the holistic twig joins algorithm to guarantee that solutions are generated. Some nodes that are located outside of the $(MostLeft, MostRight)$ positions of the ancestor's stream partition are trimmed; thus, it helps speeding the computation of the holistic twig joins. Beyond that, we notice that some nodes are possibly duplicated to maintain the range containment property between two partitions of the base stream and the ancestor's stream.

The top-down partition approach has a similar mechanism to the bottom-up approach to partition the intended stream by finding the containment property. For each partition of the descendant's stream, $(MostLeft, MostRight)$ positions are obtained by searching the descendant's stream that satisfies the range containment properties. For each partition of the ancestor's stream, its $(MostLeft, MostRight)$ positions are used to obtain $(MostLeft, MostRight)$ positions of the descendants stream by satisfying the left and right containment properties. It is often the case that partitions of the ancestor's stream contains fully duplicated XML nodes. Then, we just need to partition the descendant's stream equally according to the window size. This downward approach also has the advantage of trimming left-over stream nodes that are not included in the partitions. Figure 9 (b) shows an

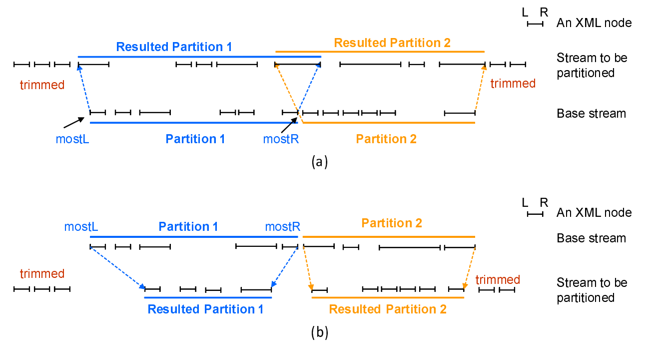

Fig. 9 An example of (a) an Bottom-Up partition, (b) a Top-Down partition.

example of the top-down partition approach.

4.3 Distributing Partitions Dynamically

The main objective is to average dynamic workloads among cluster nodes. The overall dynamic workloads are roughly estimated from accumulating processing costs of queries being executed in the system. Also, the workload average δ can be computed directly from the overall workload divided by the number of cluster nodes.

$$NP_i = \frac{\delta - WL_i}{\sum_{p=1}^{N} \delta - WL_i} \times \frac{WL_j - \delta}{WL_j} \times \#parts_j \quad (4)$$

The Eq. (4) computes the number of partitions NP_i to be distributed on an underloaded cluster node i from an overloaded cluster node j by proportioning workloads of an underloaded cluster node i over the overall workloads of underloaded cluster nodes toward the number of partitions.

Initially every time a cluster PC receives a query to be processed from the coordinator, it estimates the amount of workloads that may need to be allocated to other cluster nodes. The running query states dispatched by the coordinator are analyzed against the distribution information including query processing costs to identify which queries are running on which cluster PCs. The analysis also estimated the average workload δ of processing costs of all queries being executed, to determine which cluster PCs are underloaded or overloaded. Note that only overloaded cluster PCs computes the Eq. (4).

Subsequently, every overloaded cluster node determines an initial stream to be partitioned first, then performs Bottom-Up propagation followed by Top-Down propagation to partition the rest of the streams. Finally, resulted partitions are redistributed to underloaded cluster nodes according to Eq. 4.

5. Experimental Evaluation

The main objective of this experiment is to show the effectiveness of our proposed partitioning schemes in dealing with static XML data partition and dynamic XML data partition in order to achieve good performance of the overall query execution.

5.1 XML Data Sets and Experimental Platform

11 different XML data sets used in the experiment are derived from real data sets used for experiments in Niagara Query Engine Project and by Stanford University Infolab, and from synthetic data sets of XOO7 and XMark benchmarks. Each XML data set is characterized by the number of DTDs, the number of query twig patterns, the number of XML documents and the size of XML data set in bytes. In total, there are 194 XML documents along with 17 DTDs and 64 query twig patterns generated randomly from the given DTDs. The total size of XML documents is about 330 MB and they are varied in sizes and contents. Statistics about these XML data sets is shown in Table 1 in more details.

Table 1 XML data sets

No	XML Data	DTD	Qry	Doc	Size (bytes)
1	Bibliography	1	4	16	158,096
2	Sport Clubs	1	3	12	162,986
3	Cars	1	6	48	1,357,856
4	Departments	1	5	19	2,722,723
5	Purchases	1	2	10	4,873,260
6	Quotes	1	2	10	4,412,418
7	Dramas	1	3	18	7,428,278
8	Sigmod 2002	3	10	43	2,934,193
9	Movies	5	21	5	28,463,633
10	Auction	1	4	8	119,900,420
11	Assembly	1	4	5	171,309,031
	Total	17	64	194	343,722,894
	Average				1,771,767
	Variance				6.20E+13

Table 2 Statistics of XML data partition results

1	Document dimension size	194
2	Query-path dimension size	177
3	Query dimension size	64
4	Finest grid instances	2,551
5	The number of distinct streams	485
6	The number of XML nodes	20,185,704
7	The number of duplicated streams	61
8	The number of duplicated XML nodes	1,873,233

The experimental platform used is a shared-nothing homogeneous cluster system. One node plays a role as the coordinator and 8 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 between the coordinator node and the processing nodes.

5.2 Evaluation of Static XML Data Partition

In this subsection, we evaluate the results of our proposed partitioning scheme statistically according to GMX computation and parallel system performance.

5.2.1 Statistics of XML Data Partition

In the GMX construction, the sizes of the document dimension and the query-path dimension in its structure were 194 and 177, respectively. The GMX structure maintained 2,551 finest grid instances and required small storage to store the sparse data. Detail statistics about the total number of XML elements and their duplication is shown on Table 2.

We examined the query distribution resulted from

Table 3 Distribution of estimated workloads and queries

Cluster Node Id	Initial Workloads	Final Workloads	Final Query Distribution
1	2,267.12	1,937.23	Q5, Q6, Q33, Q34, Q35, Q43, Q64
2	10,409.22	2,829.01	Q33
3	845.77	2,184.84	Q18, Q19, Q27, Q28, Q29, Q30, Q31, Q32, Q33, Q36, Q45, Q55, Q56, Q57, Q58, Q59, Q60, Q61, Q62, Q63
4	881.35	2,040.94	Q1, Q2, Q5, Q6, Q7, Q8, Q9, Q10, Q36, Q40, Q42, Q52
5	680.73	2,278.75	Q15, Q16, Q22, Q23, Q25, Q27, Q28, Q29, Q30, Q31, Q32, Q33, Q34, Q35, Q46, Q50
6	801.10	2,170.93	Q7, Q17, Q20, Q21, Q24, Q33, Q36, Q64
7	880.55	2,355.09	Q3, Q4, Q11, Q12, Q13, Q14, Q36
8	973.17	1,942.22	Q11, Q12, Q14, Q26, Q36, Q37, Q38, Q39, Q41, Q44, Q47, Q48, Q49, Q51, Q52, Q53, Q54
Total	17,739.01	17,739.01	
Threshold τ	2,217.38	2,217.38	
Variance	1.12E+07	8.36E+04	

distributing partitions. As shown in Table 3, 18 long-running queries that accessed considerably large XML documents were potentially executed by several cluster nodes for intra query parallelism execution. For example, a query Q33 was distributed to cluster node id 1, 2, 3, 5, and 6. The rest of 46 queries were executed in a single cluster node for inter query parallelism.

5.2.2 Parallel System Performance

In this subsection, we evaluate workload balance in all cluster nodes and the parallel speed up performance based on the entire query execution. To the best of our knowledge, XML data partitioning method specifically for the holistic twig joins processing does not exist yet. Here, we compare the grid metadata partitioning method (GMX) with two basic partitioning methods: a document-based partitioning method (DOC) and a document clustering-based partitioning method (DCLUS).

The XML data partition and distribution schemes of the two basic methods are described as follows. In DOC, the basic unit of partition was an XML document where its size was considered as the cost of a partition. We distributed XML documents with Round Robin approach followed by the heuristic rule of redistributing partitions. As the result, queries associated with certain XML documents were potentially executed according to the distribution of the XML documents onto cluster nodes.

In DCLUS, we clustered XML documents with the same technique as ours where a cluster was regarded as the basic unit of partition. A cost of a partition was derived from accumulated sizes of XML documents residing in a cluster. Similar to ours, the distribution method adopted Round Robin approach combined with the two heuristic rules. In this case, refining a partition was to compute its subclusters of XML documents. Queries associated with certain XML documents in a partition were distributed according to the distribution of XML document partitions.

In the first experiment to measure the quality of workloads, we used the relative load imbalance index (LI) [15], given by this formula

$$LI = f(x_1, x_2, ..., x_N) = 1 - \frac{\sum_{i=1}^{N} x_i}{N \cdot max_{i=1}^{N} x_i} \qquad (5)$$

where x_i is a workload measure at a cluster node P_i. The index is bounded in the range [0, 1]; a lower value means better workload balancing. In this case, the workload of P_i is the time required to execute all queries residing in P_i.

Figure 10 shows the quality of workload balancing. The workload balance of GMX is superior to the other methods as indicated by lower values of load imbalance indices for all processing nodes. The two basic methods DOC and DCLUS suffered from workload imbalance because they were lack of partition refinement methods when encountering such a large XML document. However, generally DCLUST showed smaller workload imbalance indices because it has a partition refinement mechanism by computing subclusters. As for the load imbalance index of GMX, GMX suffered from the load imbalance at 8 cluster nodes where a partition with an extremely high cost could not be refined further by our partitioning refinement methods.

The next experiment, the speed up performance, or equivalently the running time, gives an immediate measure of the effectiveness of different XML data partitioning methods. In the experiment, the coordinator

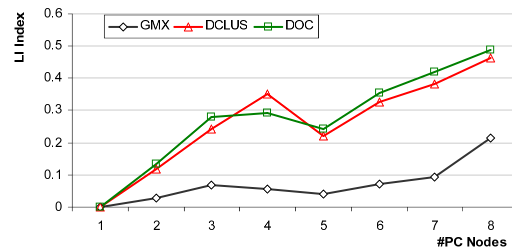

Fig. 10 Load imbalance index.

generated a random sequence of all 64 queries and distributed them simultaneously to processing nodes. We measured the total execution time of the 64 queries starting from query distribution to the last solutions received by the coordinator.

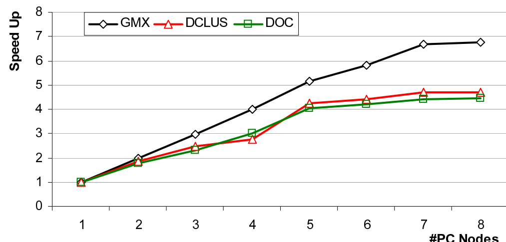

Fig. 11 Speed up performance.

Figure 11 shows the speed up performance for each partitioning method. It confirms that the GMX speed up outperformed all the others. Due to its superior workload balance, GMX was able to minimize the idle time of cluster nodes and, eventually, it contributed better speed up performance with almost linear escalation. On the other hand, the speed up performances of DOC and DCLUS were saturated starting from 5 cluster nodes. However, DCLUST had slightly better speed up performance than DOC since clusters of XML documents implied more efficient execution of similar queries allocated in the same cluster nodes.

Table 4 Queries over XMark data set

Query	XPath Expression
Q1 - Simple Query	//person[/name]/emailaddress
Q2 - Complex Query	//site//open_acution[/type/'Regular'] [/quantity/'3']/anno-tation[/description] /happiness/'4'

5.3 Evaluation of Dynamic XML Data Partition

In this subsection we evaluate statistics of streams-based partition method and the improvement of parallel system performance.

5.3.1 Statistics of XML Data Partition

The main objective is to show the efficiency of our proposed streams-based partition scheme. We adopted XML Benchmark Project as our experiment data set. We used different sizes of XML data 110MB, 330MB, and 550MB. Two query twig patterns with simple and complex structures were adopted as shown in Table 4. The total stream sizes required to execute query 1 and query 2 for 110MB, 330Mb and 550MB are given in Table 5.

The first test is to determine the best window size

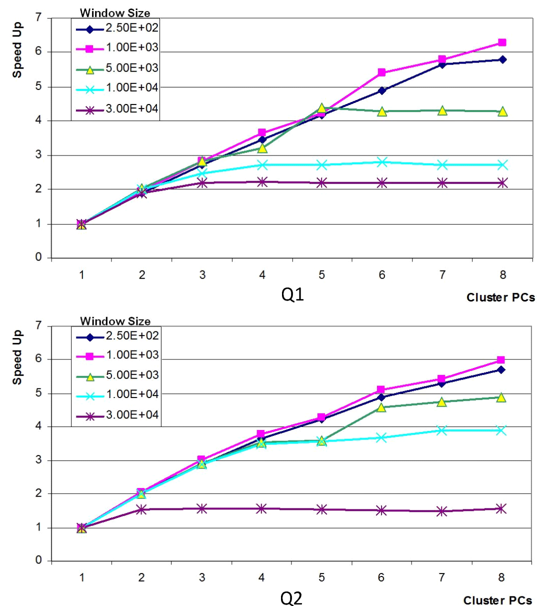

Fig. 12 Speed up performance contributed by different window sizes for 110MB XML data size

for initial partition. We can see in Figure 12 that a windows size of 1,000 gives the best speed up performance for both queries. A smaller window size introduces more overheads in forming candidate solutions of twig queries. It is shown by the window size of 250 that gives slightly lower speed up performance. On the other hand, a larger window size causes workload imbalance that obviously gives an immediate impact on the performance. We can see that the speed up curve for largest window size of 30,000 shows the worst performance.

Secondly, Table 5 describes the size of stream nodes for queries Q1 and Q2 resulted from conducting the partition plan with a window size of 1,000. It shows that the partition method is able to reduce partially unnecessary stream nodes, which do not give solutions, and to duplicate necessarily XML nodes to form candidate solutions. In fact, reduction of the number of stream nodes relies on the distribution of XML nodes over streams. According to the Bottom-Up partition approach, the smaller window size gives more refined partitions and produces more duplicated stream nodes.

Figure 13 illustrates the processing time for computing partitions and redistribution (communication) for queries Q1 and Q2. The generating partitions requires about constant time to compute, regardless the number of cluster PCs. The redistribution requires much more time than the partition plan and the time increases as the number of PCs are increased. The communication time including synchronization and data packing for transferring partition data contributes the biggest time in the redistribution. The computation time for 550MB data size requires less than 2.5 times

Table 5 The number of nodes as the result of conducting partition plan

Query	Data Size	Initial nodes	Duplicated Nodes	Trimmed Nodes	Processed Nodes
Q1	110MB	99,250	47	644	98,653
Q2	110MB	181,268	306	51,795	129,779
Q1	330MB	297,750	141	1,666	296,225
Q2	330MB	543,804	929	50,529	494,204
Q1	550MB	496,250	240	259	496,231
Q2	550MB	906,340	1,552	50,928	856,964

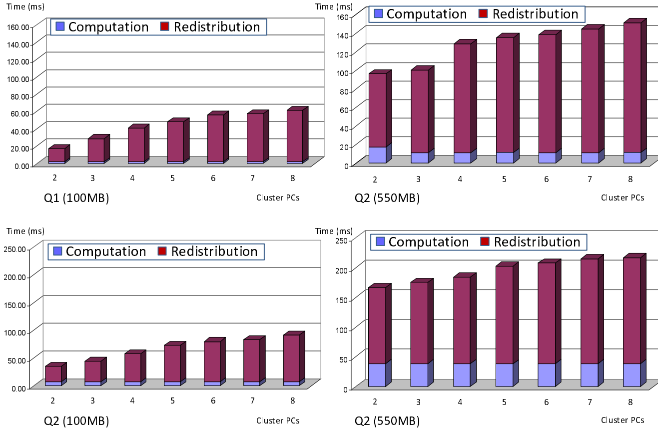

Fig. 13 The processing time (ms) to compute partitions and redistribute them for 110MB and 550MB XML data sizes

of the computation for 110MB data size. In overall, our proposed method requires minimal time and it is suitable for on-the-fly execution.

5.3.2 Improvement of Parallel System Performance

In this subsection, we show the improvement of load imbalance index, parallel speed up, and efficiency by comparing GMX (static XML data partition) and streams-based partition method.

Figure 14 shows the comparison of load imbalance index between GMX and streams-based method (Streams for short). Streams indicates better quality of load balancing on 7 cluster PCs and 8 cluster PCs because it provides partitioning method with finer granularity.

Better workload balance implies lower idle time that gives impact on better parallel speed up performance. As shown in Figure 15, the Streams speed up outperforms the GMX speed up. GMX speed up curve is no longer escalated starting from 7 cluster PCs, while GSX speed up curve indicates higher escalation.

In addition, we measure the performance improvement if only few queries are being executed in the system. We intentionally select only 5 queries to be executed in the system to simulate the imbalance workload situation as shown in Table 6.

Figure 16 shows that GMX performance is deteriorated as the number cluster PCs are increased, because of imbalance workloads during query execution. On the other hand, Streams speed up performs well because it

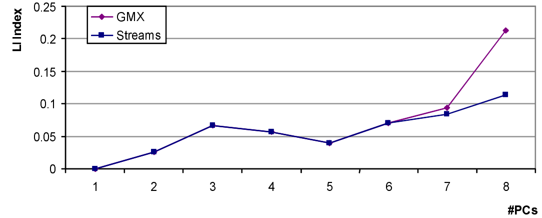

Fig. 14 Load imbalance index for static and dynamic partition.

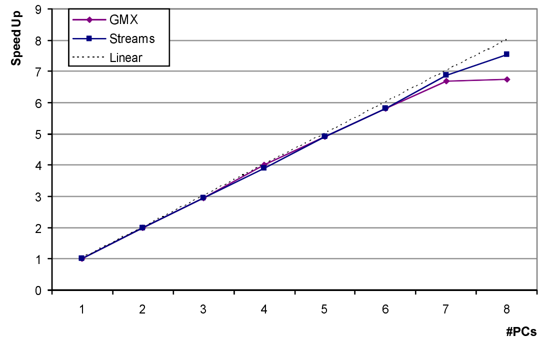

Fig. 15 Parallel speed up performance for static and dynamic partition.

Table 6 The number of nodes as the result of conducting partition plan

Query	XPath Expression	Stream Size
Q1	//article[/title/Semantic[/Web]] [//author/Wei[/Han]]//format/PDF	15,922
Q2	//company/Profile/EmployeeNumber [//state/WI]	40,000
Q3	//compositepart//atomicpart	1,110,510
Q4	//person[/name/Kang][/address[/city /Panama]/Province]/Creditcard/6284	75,182
Q5	//m/t[/Cleopatra]/a/Arthur	293,628

conducts dynamic partitioning and redistributing partitions to achieve better workload balance. In terms of efficiency shown in Figure 17, Streams maintains its efficiency value above 90%, while GMX its efficiency is extremely degraded.

6. Conclusion

In this study, we proposed novel XML data partitioning schemes for static and dynamic XML data partition, to achieve high scalability in parallel holistic twig join processing. The grid metadata model for XML provides

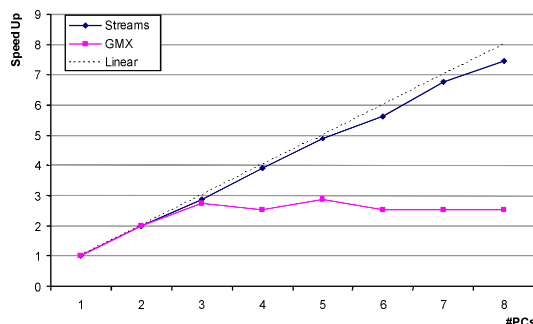

Fig. 16 Parallel speed up performance for static and dynamic partition when imbalance occurs.

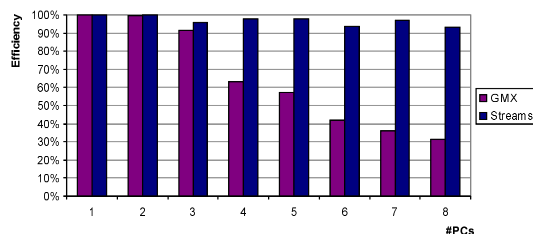

Fig. 17 Efficiency performance for static and dynamic partition when imbalance occurs.

the underlying XML data partition for static distribution, while the streams-based partition provides finer XML data partition for dynamic distribution. In the experiments, the results show the effectiveness of the proposed schemes. Distribution based on GMX methods yields balanced workloads on cluster nodes and gives impact on good parallel speed up performance when the entire queries are executed simultaneously. However, when only few queries are executed, the system performs much worse. By applying dynamic partition and redistribution the experiment shows significant performance improvement in terms both parallel speed up and efficiency.

Publication

We published 4 research papers in [12], [13], [11], [14] for this study.

References

[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 ICDE'02*, pages 141–152, 2002.

[2] T. Amagasa, K. Kido, and H. Kitagawa. Querying XML Data Using PC Cluster System. In *Proceedings of the International Workshops on XML Data Management Tools and Techniques (XANTEC'07)*, pages 5–9, 2007.

[3] J.-M. Bremer and M. Gertz. On distributing XML Repositories. In *Proceedings of the International Workshop on the WebDB'03*, pages 73–78, 2003.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 21st ACM SIGMOD'02*, pages 310–321, 2002.

[5] T. Chen, J. Lu, and T. Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *Proceedings of the 24th ACM SIGMOD'05*, pages 455–466, 2005.

[6] G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. In *IEEE Transactions on Knowledge and Data Engineering*, volume 19(10), pages 1381–1403, 2007.

[7] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *Proceedings of the 23rd ACM SIGMOD'04*, pages 779–790, 2004.

[8] K. Kido, T. Amagasa, and H. Kitagawa. Processing XPath Queries in PC-Clusters Using XML Data Partitioning. In *Proceedings of the 22nd ICDE Workshops*, pages 114–119, 2006.

[9] H. Kurita, K. Hatano, J. Miyazaki, and S. Uemura. Query Processing for Large XML Data in Distributed Environments. In *Proceedings of the 21st International Conference on AINA07*, 2007.

[10] J. Lu, T. Chen, and T. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. In *Proceedings of 13th International Conference on CIKM'04*, pages 533–542, 2004.

[11] I. Machdi, T. Amagasa, and H. Kitagawa. An Algorithm for Parallel Holistic Twig Joins on a PC Cluster. In *Proceedings of the International Database Forum (IDB'08)*, pages 1–6, 2008.

[12] I. Machdi, T. Amagasa, and H. Kitagawa. Cube-based Analysis for Maintaining XML Data Partition for Holistic Twig Joins. In *Journal of the Database Society of Japan (DBSJ)*, volume 7(1), pages 121–126, 2008.

[13] I. Machdi, T. Amagasa, and H. Kitagawa. Gmx: An XML Data Partitioning Scheme for Holistic Twig Joins. In *Proceedings of the 10th International Conference on iiWAS08*, pages 137–146, 2008.

[14] I. Machdi, T. Amagasa, and H. Kitagawa. XML Data Partitioning Strategies to Improve Parallelism in Parallel Holistic Twig Joins. In *Proceedings of the 3rd International Conference on ICUIMC'09*, 2009.

[15] 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.

[16] N. Tang, G. Wang, X. J. Yu, K.-F. Wong, and G. Yu. WIN: An Efficient Data Placement Strategy for Parallel XML Databases. In *Proceedings 11th International Conference on ICPADS'05*, pages 249–355, 2005.

[17] H. Wang, S. Park, and P. Yu. Vist: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the 22nd ACM SIGMOD'03*, pages 110–121, 2003.

[18] 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 20th ACM SIGMOD'01*, pages 425–436, 2001.