1 Introduction

Data partitioning strategy was originally studied in a shared-memory environment, where XML data is stored in a shared memory and can be concurrently accessible by multiple XPath processors. In the conclusion of the original paper [?], the authors had pointed out that the parallelization model is over XML data model, and it can also be adapted to any XML storage laryout. However, no matter in the original study, or in our previous study, the strategies are applied both in a shared-memory environment. Therefore, here comes a question: how we can apply it in a distributed-memory environment? In this study, by exploiting horizontal fragmentation on XML data, we present our study on applying data partitioning in a distributed-memory environment to experimentally show how it improves the scalability.

2 Fragmentation

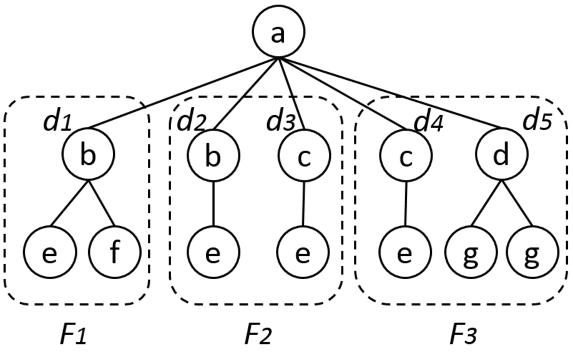
2.1 Introduction

Fragmentation is an effective way to improve scalability of database systems [?, ?, ?]. In the field of parallel XML processing, there are also some studies on fragmentation of XML data [?, ?]. The most common XML fragmentations are horizontal fragmentation and vertical fragmentation [?]. Due to the nature of horizontal fragments that are relatively independent, it is a more direct and practical way to work together with data partitioning. We thus focus on only horizontal fragmentation in this study.

2.2 Definitions

We first define horizontal fragmentation. Let $D = \{d_1, d_2, ..., d_n\}$ be a collection of document trees such that each $d_i \in D$ conforms the same XML schema [?]. Let $FS = \{F_1, F_2,..., F_m\}$ be a collection of fragments such that for each $F_i \subset D$. If $\bigcup\limits_{i=1}^{m} F_i = F$ and $\bigcap\limits_{i=1}^{m} F_i = \emptyset$, then FS is a horizontal fragmentation of D.

Let us take the tree in Fig. 1 as an example. There are five document trees in Fig. 1(a), i.e. we have $D = \{d_1, d_2, d_3, d_4, d_5\}$, where d_1 is the first subtree rooted at b (from left to right), d_2 be the second and so on. All the subtrees follow the schema in Fig. 1(b). We can make three horizontal fragments $FS = \{F_1, F_2, F_3\}$ as shown in the three dotted rectangles, where $F_1 = \{d_1, d_2\}$, $F_2 = \{d_3\}$ and $F_3 = \{d_4, d_5\}$.



(a) An example of horizontal fragmentation

schema

a(b*, c*, d)
b(e, f?)
c(e)
d(g*)

(b) The schema of the example

Figure 1: An example of horizontal fragmentation and the schema.

2.3 Applying Fragmentation

In our study, we first apply horizontal fragmentation to the input XML data, through which an XML tree is divided into a set of fragments. To make the fragmentation works well in combination with data partitioning strategy. We have the following requirements.

2.3.1 Fragments are size-balanced

Since the mainpurpose of fragmentation is to achieve good scalability, we attempt to make our fragmentation algorithm size-balanced, i.e. to make each fragment have nearly the same amount of node.

2.3.2 Path to the root is added

In our design desire of fragmentation algorithm, we also intent to ease the querying. Thus, we require that each fragment is a set of consecutive sibling subtrees augmented with the path to the root. Note that two subtrees are sibling subtrees if the roots of the subtrees are siblings of each other.

2.3.3 Correctness

We assume that all the results lie in the subtrees on the fragment. Thus, to guarantee the correctness of query results, we need to guarantee the completeness and uniqueness of nodes such that each node in the input XML tree is included in at least a fragment. If a node is included a subtree part of a fragment, then it is not included in any other fragment.

Algorithm 1 FRAGMENTATION(*nodes*, *MAXSIZE*)

```
Input: nodes: a list of nodes,
       MAXSIZE : the maximum number of nodes in a fragment
Output: a list of fragments
fragments ← [] //a list of fragments
subtrees ← [] //a list of root nodes of subtrees
for all $node \in nodes$ do
    if size(node) > MAXSIZE then
        if $subtrees.length > 0$ then
            fragments.Add((_, subtrees))
            subtrees ← []
        end if
        fragments.AddAll(Fragmentation(child(node), MAXSIZE))
    else if size(node) + Size(subtrees) > MAXSIZE then
        fragments.Add((_, subtrees))
        subtrees ← [node]
    else
        subtrees.Add(node)
    end if
end for
for $i \in [0, fragments.length)$ do
    fragments[i] ←AddPath(fragments[i])
end for
return fragments
```

Figure 2: The fragmentation algorithm.

```
Algorithm 2 AddPath(subtrees)
Input: subtrees: a list of root nodes of subtrees
Output: the root node of a tree that is augmented with the
        path to the root of the whole tree
p ←parent(subtrees[0])
node ← clone(p)
node.addChildren(subtrees)
while parent(p) ≠ NULL do
    p ← parent(p)
    tempnode ← clone(p)
    tempnode.addChild(node)
    node ← tempnode
end while
return node
```

Figure 3: Add path to a list of subtrees

Figure 4: An example tree with number denoting PRE values of nodes

Figure 5: Recontructed fragments.

Figure 6: Regroup and merge.

Algorithm 1 describes how our fragmentation works to apply a horizontal fragmentation to a tree. The arguments of input are a list of nodes denoting the tree to be fragmented and an integer number denoting the maximum number of nodes a fragment can have so that we can make the fragments in similar size. In Line 1–2, we declare an empty list of fragments and an empty list of subtrees for holding results. In Line 3–16, we traverse each node in the input list, if the node have more number of descendant nodes greater than MAXSIZE, we apply the fragmentation on the children of the node (Line 4–9) and add the results into $fragments$. Else if total number of descendant nodes in *subtrees* and the *node* excesses MAXSIZE, we save the current fragment and put the current node into a new fragment (Line 10–12). Otherwise, the current node is added to *subtrees* as one of the subtrees in the current fragment. After the iteration, we obtain a list of fragments, each of which is a list of subtrees. We add the last use Algorithms 2 to complete each fragment by adding the path from the current subtrees to the root of the original tree (Line 17–19). In Algorithm 2, we basically keep looking upward, to add all the ancestor nodes to the current fragment.

There are also several functions used in Algorithms 1 and 2 that are described as below:

- `size`(*node*) returns the number of descendants of *node*, where node is a single node.
- `Size`(*nodes*) returns the sum of number of descendants of each node in *nodes*, where *nodes* is a list of nodes.
- `child`(*node*) returns the children of *node*. - `parent`(*node*) returns the parent of *node*.
- `clone`(*node*) returns a node cloned from *node*. The function create an empty node and copy the name and attributes from *node*.

Note that for the query processing, we need a representative value for each fragment to maintain the original order of a fragment as in the original tree. We simply use a fragment id for each fragment.

Let us take the tree shown in Fig. 4 as an example with MAXSIZE = 3. After Line 16 of Algorithm 1, the fragments are list of subtrees. When we use the PRE index to denote the root node of a subtree, we have F = [f$_1$, f$_2$, ..., f$_8$] = [[2, 5], [7], [12], [15, 18], [21], [24], [28], [30]], where the subscripted numbers are the fragment ids. Then, by applying Algorithm 2 that adds the path from the subtrees to the root of the whole tree, we obtain the final recontructed fragment as shown in Fig. 5.

3 Our Ditributed XPath Query Framework

We design an XPath query framework using horizontal fragmentation with data partitioning strategy on top of BaseX over a distributed-memory environment. In this framework, there are one client and N_s servers. The client is a computer running a Java program that is the implementation of our query algorithm. It works for sending queries to multiple servers and processing results returned from them. A server is a computer that runs a BaseX server in charge of evaluating received queries. An input dataset will be fragmented and distributed to all the servers to be queried and the results returned from all the servers will be merged on the client.

It consists of the following four stages:

- Data Fragmentation
  To divide an input XML document into fragments.

- Allocation
  To assign the fragments into multiple computation nodes.

- Query Evaluation
  To query fragments in parallel
- Results Merging
  To merge the results of fragents to form the final result.

We give the detailed introduction in the following sections.

3.1 Allocation

After fragmentation, the fragments are mapped to multiple computation nodes. For each computation node, a sub set of fragments is assigned. To make the distribution of fragments well-balanced, we randomly shuffle the list of fragments before dividing the list into N lists of fragments, where N is the same as the number of computation nodes.

We use a mapping list to map the fragments. An element in the mapping list contains information for locating the corresponding fragment so that we can access and process query on the fragment independently. With this mapping list, we can start a query by locating the root of each fragment on any tree, making it possible to evaluate queries on them in parallel. And also, the mapping list can be used to maintain the order of results to form the final results.

In our study, we run a single BaseX instance in server mode on each computation node. For each computation node, the sub set of fragments on it is then added to an empty root node to form a complete XML tree so that we can load the tree by the BaseX server to create an XML database for further query evaluation. To make each list of fragments become a single tree (so that we can create a database in BaseX from it), we create a node as the root and add the lists of the fragments to the root, where the root of a fragment becomes the child of the newly added root node.

Let us continue the running example. Let N be 4, after shuffling and regrouping, we may obtain four groups of fragments: FS => [F$_1$, F$_2$, F$_3$, F$_4$], where F$_1$ = [f_7, f_4], F$_2$ = [f_3, f_1], F$_3$ = [f_5, f_0], F$_4$ = [f_6, f_2]. By adding a root node vn to the each group, we create four trees from FS respectively as shown in Fig. 6.

We also create a mapping list of links $links$ for each F in FS for locating. A link is a 3-tuple $(Tree, pre, depth)$, where $tree$ is a pointer to a reconstructed tree, pre is the PRE value of the root of a fragment in $tree$, $depth$ is the number of nodes in the path from subtrees to the root. For the given example, we have a list of links $links$ = [(Tree$_3$, 2, 1), (Tree$_2$, 9, 1), (Tree$_4$, 6, 2), (Tree$_2$, 2, 2), (Tree$_1$, 8, 2), (Tree$_3$, 2, 2), (Tree$_4$, 2, 2), (Tree$_1$, 2, 2)], where $links[0] \to f_0$, $links[1] \to f_1$, ..., $links[7] \to f_7$.

4 Query evaluation

4.1 Rewriting An XPath Query In XQuery

An input XPath query is rewritten into an XQuery expression to be then processed by BaseX servers. The rewriting is different depending on whether data partitioning strategy is used or not.

4.2 Without Data Partitioninng Strategy

Since the nodes in the results will no long follow the original order in the input document, we return the nodes along with their PRE index for later identifying and reordering by using the following expression.

```
for $node in db:open('db')$query
  return ((' ', db:node-pre($node)), $node)
```
[1]

We separate the PRE value and the content of a node by a linebreak and add an extra linebreak among resultant nodes[2].

4.3 With Data Partitioning Strategy

When applying data Partitioning, we use the server-side implementation. In order to maintain the order of resultant nodes, we make some change to the server-side implementation. For the two-phases implementation, we do not need to change the first phase, which still returns the PRE values of the results of prefix query. We need to change the second phase, where the `return` statement in the suffix query needs to be modified to `return ((' ',db:node-pre($node)), $node)`, i.e. the same as the XQuery expression described in 4.2.

5 Evaluating Queries

The evaluation an XPath queries consists of two steps: sending query and processing results.

5.1 Sending query

After an input query being rewritten, it will be sent form the client to all srevers for executing. After sending, the client will be idle waiting for results to be sent back.

[1] `return (a, b)` will add a line break between `a` and `b` while returning.

[2] According to my previous implementation and tests, it worked.

5.2 Processing Results

The results are returned from all servers through the network (some servers may return empty results). There are two issues we need to consider when processing results. First, since the size of results can be larger than the memory size of the client (such as XM1 that returns about 90 the size of the input data), we thus store results on disk. Second, due to the randomization, the results returned from all servers are not in the original order. Thus, we have to recover the original order of results when processing. It is different to deal with the order depending on whether data partitioninng strategy is used.

5.3 Without Data Partitioning Strategy

First, since we have the PRE values of resultant nodes, we can use them to determine which fragment a resultant node belongs to by comparing with the *mpre* of each fragment, where *mpre* is the PRE value of the root of the first subtree in the fragment. Given a list of fragments $F = \{f_0, f_1,..., f_n\}$, a function $\textsc{GetMpre}(f)$ that returns the *mpre* of the fragment f and the PRE value *pre* of a node n, if $\textsc{GetMpre}(f_i) \leq pre < \textsc{GetMpre}(f_{i+1})$, ($i < n$ - 1) or $\textsc{GetMpre}(f_i) \leq pre$, ($i = n$ - 1), then n belongs to f_i. For example, there are three fragments, and their *mpre* are 3, 10, 20 respectively. A node with the PRE value 5 belongs to the first fragment. Note that we do not need to deal with the order of the nodes in the same fragment, because these nodes in the same fragment still keep the original order. Thus, the results can be grouped by fragments.

Next, we store the results in a list of files, each of which stores the results that belong to the same fragment. We use fragment id to name these files so that we can obtain the results stored in these files in the original order, when reading the files ordered by their names

Through the above method, since all the fragment can be processed separately, we can receive the results and save them to disk in parallel.

5.4 With Data Partitioninng Strategy

Since data partitioninng strategy uses multiple processors, results of the same fragment may be processed by two processors. For example, when we use two processes p_1 and p_2 to process the same merged tree with only one fragment f_0, the resultant nodes in the fragment may be processed by both processors. Since there is only one file that corresponds the fragment for storing, the two processor cannot write the resutls to the file at the same time.

There are two ways to solve the problem. One way is to write the results in sequential, i.e. we let only one processor to write at a time. For example, we let p_1 write first then p_2 follows. However, this way should be slow, because when a processor is writing results, the following processors have to wait. Another way is simply to add a suffix to the file names then concatenate files with suffix. For the above example, p_0 changes the file name to `0_start.txt` and p_1 to `0_end.txt` for the fragment f_0. After receiving all the results, we concatenate the two files to `0.txt`. In this way, we can still process the results in parallel. But in some extreme case when the results of a fragment is very larger, the concatenation may bring significant overhead.

6 Experiments

For the experiment part, I decide to use 4 computers, each has 4 cores (matsu-lab50–matsu-lab53). The XMark dataset sizes 4.4 GB with 66.65M nodes. The MAXSIZE is set to 4.2M, so that we can divide the dataset into 16 fragments (my intention is to assign one to each core). I use two settings for experiments. One that only runs queries in parallel and the another also applies to data partitioning strategy. We will compare the two setting to show how much DPS can be used to improve the query efficiency.