Journal of Information Processing Vol.24 No.2 425–438 (Mar. 2016)

[DOI: 10.2197/ipsjjip.24.425]

Regular Paper

A Partial-tree-based Approach for XPath Query on Large XML Trees

Wei Hao[1,2,a)] Kiminori Matsuzaki[1,b)]

Received: July 3, 2015, Accepted: October 27, 2015

Abstract: XML is a popular data definition language and is widely used for representation of arbitrary data structures. For queries on XML documents, XPath has commonly been used in many applications. The complexity of applying queries increases as the number of nodes in an XML document increases. Querying very large XML documents becomes really difficult when there is not enough computer memory to store and manipulate the whole tree data. The objective of this study is to develop an algorithm for querying very large XML trees in a distributed-memory environment. We split a large XML document into small chunks and parse the chunks to create special trees called partial trees. Then the query is executed in parallel on the partial trees. The results from the partial trees are concatenated to form the final query results for output. The algorithms were tested on a 16-node PC cluster, and the experiment results showed a speedup of a factor of 6 on 16 nodes.

Keywords: XML, XPath, XML query, parallel programming, partial tree

1. Introduction

XML [21] (eXtensible Markup Language) is a standard language for organizing and representing semi-structured (tree-structured) data, and it has been widely used for decades. The success of XML has led to a great many applications that have been specially developed for XML. Among them, XPath [22] is a query expression language for XML, and it uses a path expression to specify a set of XML elements. XPath is also the basis of other query languages such as XQuery.

In the last decade, the rapid growth of the amount of information has led to an urgent demand for high-performance data processing technologies for business and scientific research. When the size of XML data exceeds the size we can deal with by conventional DOM-based tools, we need more involved techniques such as parallelization in distributed-memory environments or stream processing.

When we compute in parallel in distributed-memory environments, we first need to divide the input into smaller parts and allocate them to the computers. One possible approach is to adopt a tree-dividing technique for the tree that an XML document represents. A naive way is to divide a tree at the root or at a fixed depth, but this does not guarantee the size of subtrees. A more involved way is to apply the m-bridge technique [9], [14] with which we can divide a tree into parts no larger than the parameter m. However, these tree-based divisions require parsing the whole XML document in advance, and this may limit the applications.

[1] School of Information, Kochi University of Technology, Kami, Kochi 782–8502, Japan
[2] Department of Computer Science and Engineering, Anhui University of Science and Technology, Huainan, Anhui, China
[a)] 188004h@gs.kochi-tech.ac.jp
[b)] matsuzaki.kiminori@kochi-tech.ac.jp

In this paper, we propose another approach for input division in which we divide the XML document (text). Usually, XML data are stored in the serialized format, and it is very easy to divide a text into smaller chunks. It is, however, not trivial to apply queries for those chunks because some necessary information to applying queries is missing in a chunk.

To clarify the problem, consider that the input is the following XML document and a chunk is given from the underlined part.

```
<A><B><C>c1</C><C>c2</C><C>c3</C><A><C>c4</C><B>
</B></A></B><A><B></B></A><B></B></A>
```

Figure 1 shows the tree structure that the XML document represents. Here comes a fundamental question. *What structure does the chunk represent*? The chunk includes tags (beginning and/or end tags), which are gray in Fig. 1. Note that some tags, such as the first </C> or the last <B>, miss their matching tags. It seems that we cannot obtain the structure and that it is impossible to apply the query to a randomly split chunk.

To solve this problem, we add some nodes from the root of the tree and formalize the idea as a *partial tree* as shown in **Fig. 2** (the figure has four different types of nodes, which will be dis-

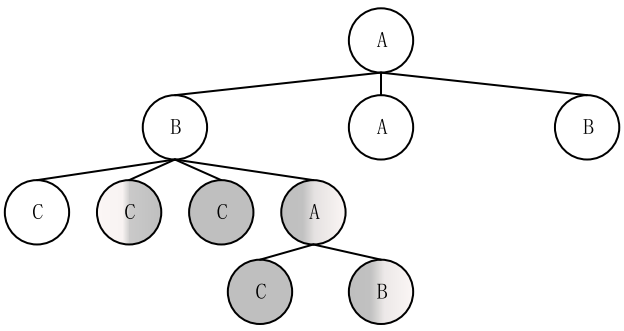

Fig. 1 An example XML tree.

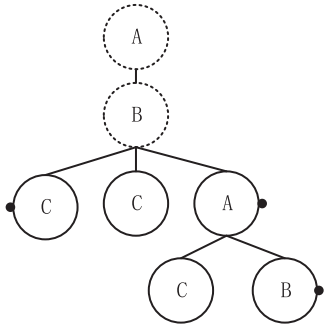

Fig. 2 Partial tree for the example.

cussed in detail in Section 3). By adding the nodes on the path from the root, we are now able to apply the queries based on the parent-child relationships.

In this paper, we deal with an important subset of XPath queries called *navigational XPath queries* [7] in which we specify the nodes with not only the parent-child relationships but also the intra-sibling relationships and additional conditions called predicates. Although a partial tree has the parent-child relationships from the root, it lacks the information required to process navigational XPath queries: a node may have only some of its children, and some siblings may be on another partial tree. Therefore, we develop a new algorithm for processing navigational XPath queries over a set of partial trees with communication. Our algorithm is easy to implement because it processes the steps in a query (including those in predicates) one by one and is efficient because we carefully analyzed the conditions to reduce the communication among partial trees.

The contributions of our study are summarized as follows.

- **Formalizing the structure for XML chunks**: We first formalize the structure and properties of partial trees that are given from chunks of an XML document (Section 3). We also show an algorithm to parse the chunks and construct the partial trees (Section 4).
- **Parallelizing XPath Queries**: We then develop an algorithm for executing the navigational XPath queries in parallel (Section 5). Basically, the algorithm runs independently on partial trees, but it also performs communication to obtain the correct query results.
- **Experiments on GB-level XML documents**: We implemented the algorithm in Java and conducted experiments on a PC cluster with GB-level XML documents (Section 6). Our implementation successfully processed an 8 GB XML document in parallel and obtained speedups of a factor of 6.0 over 16 PCs.

The remainder of the paper is organized as follows. In Section 2, we review the XPath query. In Section 3, we discuss the partial trees in detail. In Section 4, we discuss how to construct partial trees from chunks of an XML document. In Section 5, we discuss executing XPath query algorithms in parallel. We report the experiment results in Section 6. Related work is shown in Section 7, and we conclude the paper in Section 8.

2. XPath Query

XML path language (XPath) [22] is a W3C standard for representing queries to XML documents. In XPath, a query is represented in path notation. In this paper, we focus on an important subset of XPath queries called *navigational XPath queries* [7]. **Figure 3** shows the grammar of the XPath queries we use in this paper.

Query ::= '/' *LocationPath*
LocationPath ::= *Step* | *Step* '/' *LocationPath*
Step ::= *AxisName* '::' *NameTest Predicate*?
AxisName ::= 'self' | 'child' | 'parent'
| 'descendant' | 'ancestor'
| 'descendant-or-self' | 'ancestor-or-self'
| 'following-sibling' | 'preceding-sibling'
NameTest ::= '*' | *string*
Predicate ::= '[' *SimpleLocationPath* ']'
SimpleLocationPath ::= *SimpleStep*
| *SimpleStep* '/' *SimpleLocationPath*
SimpleStep ::= *AxisName* '::' *NameTest*

Fig. 3 Grammars of XPath queries used in this paper.

An XPath query in this paper starts from the root and consists of one or more *steps*. Each step consists of an *axis*, a *name test*, and at most one *predicate*. An axis defines a set of nodes relative to the nodes matched for the steps so far. We can use nine axes [*1], including `following-sibling` and `preceding-sibling`. A name test is used for selecting nodes: if the name of a tag in an XML document is equal to the name test, the node is selected. A predicate written between "`[`" and "`]`" describes additional conditions on the matched nodes by using a path without predicates. For example, "`/descendant::a/child::b[following-sibling::d]`" is an XPath query with two steps where `descendant` and `child` are the axes, `a` and `b` are the name test, and a predicate `following-sibling::d` is attached to the second step. This query first retrieves all the nodes with name `a` in an XML document, and then among their children it retrieves node `b` with one or more following siblings with name `d`. In other words, the result of the query is a set of nodes `b` that has its parent `a` and at least one sibling `d` on its right.

3. Partial Tree

The main idea of our approach for evaluating large XML documents is to split an XML document into chunks and query the chunks on different computers of a cluster. To support this, we first define the structure for presenting a chunk in the memory, which is called a *partial tree*. The partial tree is the core concept in our research. We use partial trees to represent chunks of an XML document and the XPath queries are also applied to partial trees. Therefore, to begin with, we will give a detailed introduction to the partial tree.

3.1 Node Types and Definitions

Partial trees contain many different types of XML nodes. Four types of nodes are shown in **Fig. 4**. A closed node has both its start tag and end tag. A node without one of its tags is called an open node. A left-open node is missing its start tag, and a right-open node is missing its end tag. In the figures, the missing tags

[*1] Since the name-test allows a wildcard "*", we can translate the `following` and `preceding` axes into a path in the grammar: for example `following::`x is the same as `ancestor-or-self::*/following-sibling::*/descendant-or-self::`x.

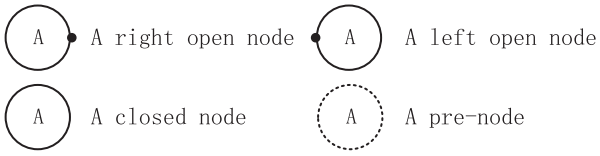

Fig. 4 Four node types.

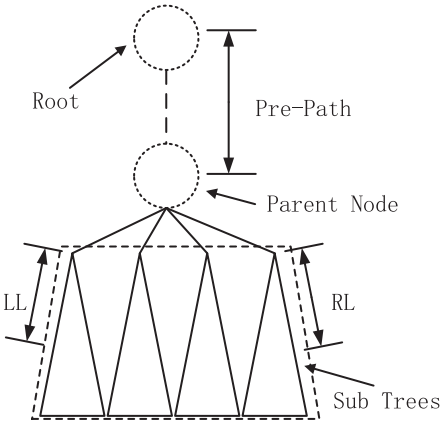

Fig. 5 A standard model of partial tree.

are illustrated by a black dot: •. A pre-open node is a node missing both of its tags. Actually a node is no longer a node if it misses both of its tags, but we need it for representing the parent node of a partial tree, which specifies the relationships from the root. Note that our research focuses on querying nodes; therefore, it is not interesting for us that no tag is contained in a chunk, even if we can denote the whole chunk as a pre-open node. All the three types of nodes, left-open node, right-open node, or pre-node, are called *open nodes*.

3.2 Standard Model for Partial Tree

Now we discuss the characteristics that partial trees generally have. Since the left-open nodes, right-open nodes, and pre-nodes are the special nodes in partial trees, we focus on the properties of the open node.

The first property is about the parent-child relationship of the open nodes.

Property 1 *If a node on a partial tree is left/right open, then its parent is also left/right open.*

The second property is about the sibling relationship of the open nodes.

Property 2 *If a node is left open, it is the first node among its siblings in the partial tree. If a node is right open, it is the last node among its siblings in the partial tree.*

There is another important property of pre-nodes.

Property 3 *If there exist multiple pre-nodes, then only one of them has left-open/closed/right-open nodes as its child.*

We develop a standard model of partial trees based on these properties as shown in **Fig. 5**.

The partial tree consists vertically of two parts: a list of pre-open nodes and a forest of subtrees. We call the list of pre-open nodes *pre-path*. The pre-path plays an important role in applying queries from the root. From property 3, one or more subtrees connect to a pre-node at the bottom of the pre-path. Note that for each subtree, there is only one root, which is a left-open/closed/right-open node, but there could be one or more subtrees.

From properties 1 and 2, we know that the left-open nodes are located on the upper-left part of a partial tree and the right-open nodes are located on the upper-right part. More precisely, the left-open nodes form a list from a root node of a subtree, and we call the list the *left list* (LL). Likewise, we call the list of right-open nodes the *right list* (RL).

4. Partial Tree Construction

Since the structure of a partial tree is different from ordinary XML trees, we designed an algorithm for partial tree construction. The algorithm for constructing partial trees has three steps: constructing subtrees from parsing chunks, pre-path computation, and computation for ranges.

4.1 Construction of Subtrees from Parsing XML Chunks

A partial tree is constructed from parsing an input XML chunk, which is a substring created from splitting an XML document. We design an algorithm that parses the input XML string into a similar tree by using an iterative function with a stack. We use an example XML document listed below to demonstrate how our algorithm works.

```
<A><B><C><E></E></C><D></D></B><E></E><B><B><D><E>
</E></D><C></C></B><C><E></E></C><D><E></E></D></B>
<E><D></D></E><B><D></D><C></C></B><B></B></A>
```

From the document, we can create an XML tree as shown in **Fig. 6**. We number these nodes in a prefix order for identification.

Then, we split the document into five chunks as listed below.

$chunk_0$:<A><B><C><E></E></C><D></D></B>
$chunk_1$:<E></E><B><B><D><E></E></D>
$chunk_2$:<C></C></B><C><E></E></C><D>
$chunk_3$:<E></E></D></B><E><D></D></E>
$chunk_4$:<B><D></D><C></C></B><B></B></A>

When splitting an XML document, we need to deal with nodes with missing tags. During parsing, we push the start tag onto the stack. When we meet an end tag, we pop the last tag to merge a closed node. However, as a result of splitting, some nodes miss their matching tags. In this case, we mark it left-open or right-open based on which part is missing. Then, we add them onto the subtrees in the same way as we add closed nodes.

We also need to handle the case when the split position falls inside a tag and thus splits the tag into two halves. In this case, we simply merge the split tags. Because there are at most two split tags on a partial tree, the time taken for merging them is negligible.

One or more subtrees can be constructed from one chunk. We construct nine subtrees by parsing the five chunks above as shown in **Fig. 7**. $Chunk_0$ and $chunk_4$ have only one subtree while $chunk_2$ has three subtrees. After the parsing phase, these subtrees are used for pre-path computation.

4.2 Pre-path Computation

The basic idea of computing the pre-paths for each partial tree is to make use of open nodes, because missing parent and ancestor nodes are caused by splitting these nodes. Therefore, the information needed for creating the pre-paths lies in these open

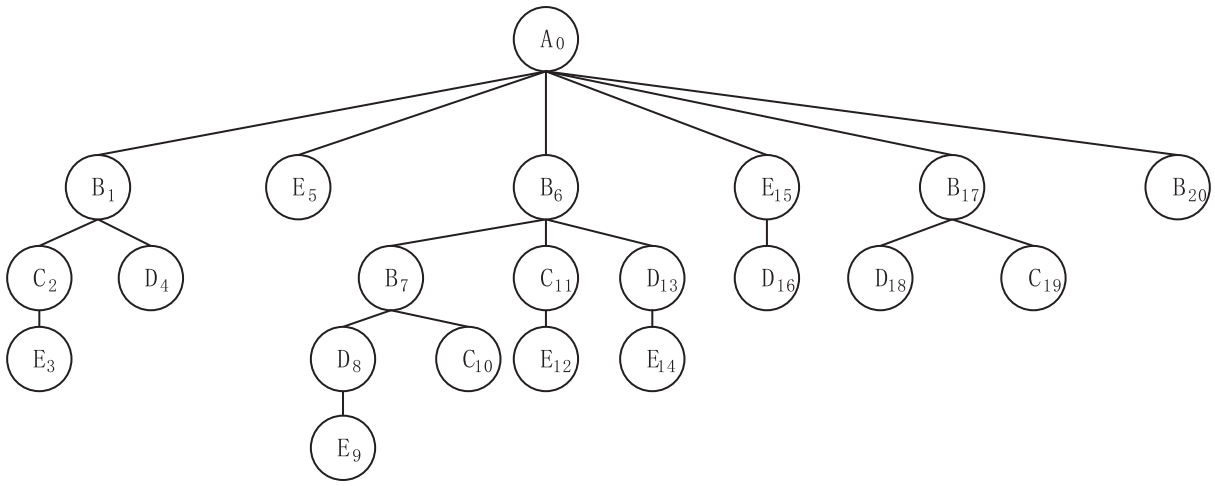

Fig. 6 An XML tree from the given XML string.

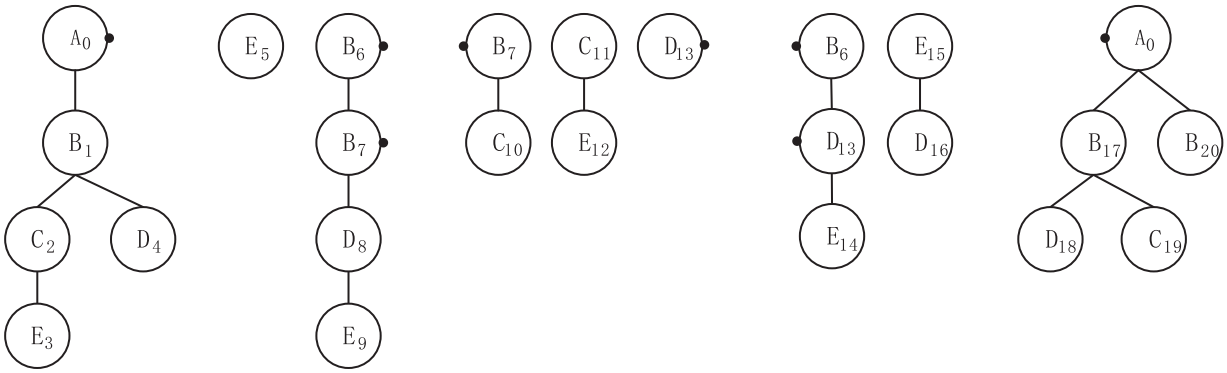

Fig. 7 Subtrees from parsing chunks.

Table 1 Open node lists.

	Left-open nodes	Right-open nodes
pt_0	[]	[A_0]
pt_1	[]	[B_6, B_7]
pt_2	[B_7]	[D_{13}]
pt_3	[B_6, D_{13}]	[]
pt_4	[A_0]	[]

nodes.

Algorithm 0 outlines the pseudo codes for pre-path computation. Because one chunk may generate more than one subtrees, the input is a list of subtree lists. The length of the list is equal to the number of partial trees, thus we represent the length as P.

Algorithm 0 has three phases. The first phase selects all left-open nodes to *LLS* and all right-open nodes to *RLS* (line 2-4). $LLS_{[P]}$ collect the left open nodes of the pth partial tree, likewise we have $RLS_{[P]}$. Note that the nodes in $LLS_{[P]}$ or $RLS_{[P]}$ are arranged in order from root to leaves. For example, in **Table 1**, we select all the open nodes and add them to corresponding lists.

In the second phase, we do the pre-path computation. Once we split an XML document from a position inside the document, the two partial tree created from splitting have the same number of open nodes on the splitting side. Given two consecutive partial trees, the number of right-open nodes of the left partial tree is the same as the number of left-open nodes of the right partial tree. We can use this feature to compute pre-paths for partial trees.

Algorithm 0 GetPrepath(*STS*)

Input: *STS*: a list of subtree lists

Output: an indexed set of partial trees

```
1:  /* open nodes in LLS or RLS are arranged in top-bottom order */
2:  for all p ∈ [0, P) do
3:      LLS_[p] ← SelectLeftOpenNodes(STS_[p])
4:      RLS_[p] ← SelectRightOpenNodes(STS_[p])
5:  /* Prepath-computation and collecting matching nodes */
6:  AuxList ← []
7:  for p ∈ [0, P − 1) do
8:      AuxList.AppendToHead(RLS_[p])
9:      AuxList.RemoveLast(LLS_[p+1].Size())
10:     PPS_[p+1] ← AuxList
11: /* Add pre-nodes to subtrees */
12: PTS ← []
13: for p ∈ [0, P) do
14:     for i ∈ [0, PPS_[p].Size() − 1) do
15:         PPS_[p][i].children.Add(PPS_[p][i+1])
16:     PPS_[p].last.children.Add(STS_[p])
17:     PTS_[p] ← PPS_[p][0]
18: return PTS
```

We first add the pth *RLS* to the head of an auxiliary list *AuxList* (line 8), then remove the same number of nodes as the number of $(p-1)$th *LLS* (line 9). Last, we keep the nodes in the *AuxList*

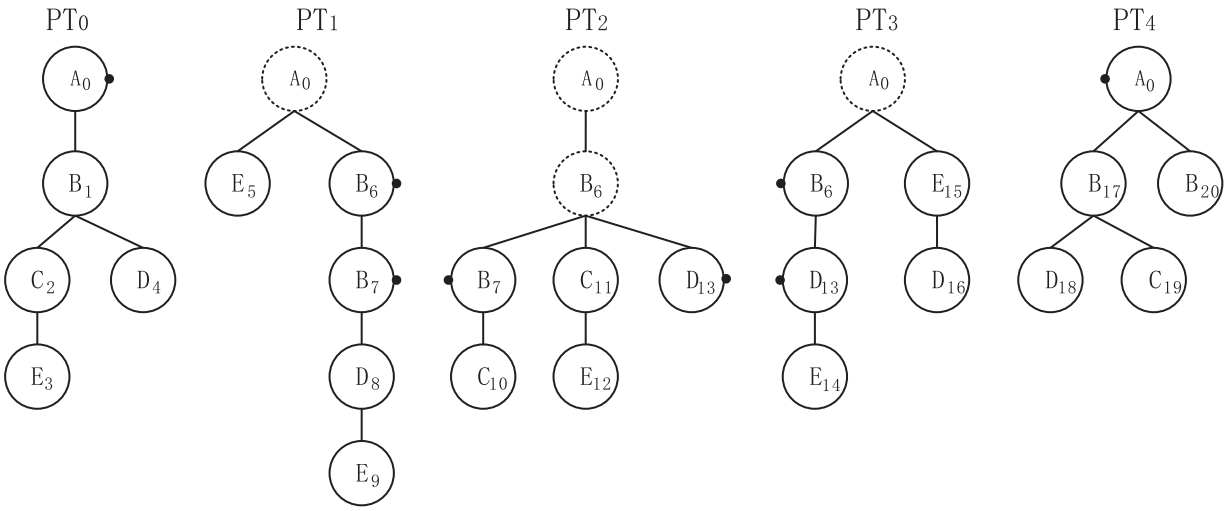

Fig. 8 Partial trees from the given XML string.

Table 2 Results of pre-path computation in AUX.

	Left-open nodes	Right-open nodes	*AUX*
pt_0	[]	[A_0]	[]
pt_1	[]	[B_6, B_7]	[A_0]
pt_2	[B_7]	[D13]	[A_0,B_6]
pt_3	[B_7, D_{13}]	[]	[A_0]
pt_4	[A_0]	[]	[]

Table 3 All open nodes.

	Left-open nodes	Right-open nodes
pt_0	[]	[A_0]
pt_1	[A_0]	[A_0, B_6, B_7]
pt_2	[A_0, B_6, B_7]	[A_0, B_6, D_{13}]
pt_3	[A_0, B_6, D_{13}]	[A_0]
pt_4	[A_0]	[]

to the $(p+1)$th PPS, which holds the pre-nodes for each partial tree. **Table 2** shows the results of pre-path computation for the given example.

In the last phase, we add pre-nodes to the corresponding partial tree and copy the nodes in $PPS_{[p]}$ to $PTS_{[p]}$ as results for output.Because pre-nodes in the pre-path are also open nodes, we list all open nodes for each partial trees in **Table 3**. Then, the pre-path computation is completed. For the given example, we obtain the partial trees as shown in **Fig. 8**.

4.3 Creation of Ranges of Open Nodes

Once an XML node is split, it generates two or more open nodes on consecutive partial trees. For example, as we can see in Fig. 8, B_6+ on pt_1, +B_6+ on pt_2, and +B_6 on pt_3 are created from the same node B_6. For locating the open nodes of the same node on different partial trees, we use two integers *start* and *end* for the open nodes. With these two integers, we can decide the partial trees that have matching nodes of the same open node. Note that after adding nodes to a partial tree, the nodes from the same node also have the same depth. Therefore, we can locate all the matching nodes to set *start* and *end* for each open node. After computation, we obtain the ranges shown in **Table 4**.

Table 4 Open node lists with ranges.

	Left open nodes	Right open nodes
pt_0	[]	[A_0(0,4)]
pt_1	[A_0(0,4)]	[A_0(0,4), B_6(1,3), B_7(1,2)]
pt_2	[A_0(0,4), B_6(1,3), B_7(1,2)]	[A_0(0,4), B_6(1,3), D_{13}(2,3)]
pt_3	[A_0(0,4), B_6(1,3), D_{13}(2,3)]	[A_0(0,4)]
pt_4	[A_0(0,4)]	[]

By using these ranges, we can locate the matching nodes of the same node on the different partial trees. For example, the range of A_0 is (0, 4), that means we can locate the same nodes of A_0 from pt_0 to pt_4. As we can see, there are A_0+, +A_0+, +A_0+, +A_0+, and A_0+ on pt_0 to pt_4, respectively.

5. XPath Queries on Partial Trees

When we design the XPath query algorithms for a set of partial trees, there are the following three main difficulties.

First, a node in the original XML tree may be split into two or more nodes in different partial trees. When such a node is selected in a partial tree (e.g., B_6+ on pt_1), the other corresponding nodes (+B_6+ on pt_2 and +B_6 on pt_3) should also be selected to be consistent.

Second, though the partial trees have all the parent-child edges of their nodes, the sibling-relation that is split among partial trees is missing. When we perform queries with `following-sibling` or `preceding-sibling`, the results may be on another (possibly far) partial tree. We need to design an algorithm to let the partial trees know about such cases.

Third, when we perform queries with a predicate, we usually execute the sub-query in the predicate from a set of matching nodes. However, on a set of partial trees, the starting nodes and the matching nodes of the sub-query may be on different partial trees. We also need an algorithm to propagate the information over partial trees for queries with predicates.

In this section, we develop an algorithm for XPath queries on a set of partial trees. We first show the outline of the algorithm and then describe the details of the queries. We use the following

three XPath expressions as our running examples.

Q1 : `/child::A/descendant::B/descendant::C/parent::B`
Q2 : `/descendant::B/following-sibling::B`
Q3 : `/descendant::B[following-sibling::B/child::C]/child::C`

We also discuss the complexity of our algorithm at the end of this section.

5.1 Node Definition

We give a few definitions to partial tree nodes for XPath queries. Each node has a *type* denoting its node type and *depth* denoting the number of edges from the node to the root. A node has four pointers pointing to related nodes: the *parent* pointer points to its parent and the *children* pointer points to its children. For accessing siblings, it has the *presib* pointer and the *folsib* pointer that point to its preceding-sibling node and following-sibling node, respectively. It is a common requirement that we should know from which partial tree a node comes in distributed memory environments; therefore, we number each partial tree with a unique id denoted as *partial tree id* or simply *ptid* for distinguishing partial trees. We number *ptid* from 0 to *P - 1* (where P is the total number of partial trees) in document order. Each node has a unique id called *uid*, and we define data type *Link* for holding ptid and uid. By using FindNode(*pt*, *uid*), we can locate any node with a unique integer uid on partial tree *pt*. We assume that we can find a node in constant time.

5.2 Queries without Predicate

Algorithm 1 in **Fig. 9** shows the outline of our XPath query algorithm. The inputs are a query and a set of partial trees. The output is a set of matching nodes, each of which is associated with the corresponding partial tree.

The query starts from the root of the XML tree. Note that the root node corresponds to the root node of every partial tree, and they are put into the lists for intermediate results (lines 1–2). Hereafter, the loops by p over $[0, P)$ are assumed to be executed in parallel.

An XPath query consists of one or more steps, and in our algorithm they are processed one by one. For each step, our algorithm calls a sub-algorithm based on its axis (given later) and updates the intermediate results (line 4).

Lines 6–9 will be executed when a query has a predicate. We will explain this part later in Section 5.3.

5.2.1 Downwards Axes

Algorithm 2 in **Fig. 10** shows the procedure for querying a step with a child axis. The input $InputList_{[P]}$ has the nodes selected up to the last step on each partial tree. The algorithm simply lists up all the children of input nodes and compares their tags with the node test (lines 3–4).

Algorithm 3 in Fig. 10 shows the procedure for querying a step with descendant axis. Starting from every node in the input, we traverse the partial tree by depth-first search with a stack. To avoid traversing the same node more than once, we add the *isChecked* flag for each node (lines 8–9). Note that we can reduce the worst-case complexity using this flag from square to linear with respect to the number of nodes.

Algorithm 1 Query($steps$, $pt_{[P]}$)

Input: *steps*: an XPath expression
$pt_{[P]}$: an indexed set of partial trees
Output: an indexed set of results of query

```
1:  for p ∈ [0, P) do
2:      ResultList_p ← { pt_p.root }
3:  for all step ∈ steps do
4:      ResultList_[P]
            ← Query⟨step.axis⟩(pt_[P], ResultList_[P], step.test)
5:      if step.predicate ≠ NULL then
6:          PResultList_[P] ← PreparePredicate(ResultList_[P])
7:          for all pstep ∈ step.predicate do
8:              PResultList_[P]
                    ← PQuery⟨step.axis⟩(pt_[P], PResultList_[P], pstep)
9:          ResultList_[P] ← ProcessPredicate(PResultList_[P])
10: return ResultList_[P]
```

Fig. 9 Overall algorithm of XPath query for partial trees.

Now let us look at our running example Q1. For the first step `child::A`, we obtain only one node for each partial tree.

pt_0	pt_1	pt_2	pt_3	pt_4
$[A_0+]$	$[+A_0+]$	$[+A_0+]$	$[+A_0+]$	$[+A_0]$

Then we perform the next step `descendant::B` independently for each partial tree from the result shown above. The following are the results up to `descendant::B`.

pt_0	pt_1	pt_2	pt_3	pt_4
$[B_1]$	$[B_6+, B_7+]$	$[+B_6+, +B_7]$	$[+B_6]$	$[B_{17}, B_{20}]$

For the third step `descendant::C`, the algorithm works in a similar way. It is worth noting that the *isChecked* flag now works. For example, on pt_1, starting from B_6+, we traverse B_7+, D_8, E_9, and then starting from B_7+, we can stop the traverse immediately. The results up to `descendant::C` are as follows.

pt_0	pt_1	pt_2	pt_3	pt_4
$[C_2]$	[ ]	$[C_{10}, C_{11}]$	[ ]	$[C_{19}]$

5.2.2 Upwards Axes

In the querying of a step with downwards axes, the algorithm has nothing specific to partial trees. This is due to Property 1 in Section 3.2. Let an open node x be selected after a downwards query. Then, it should have started from an open node (this is an ancestor of x) and the corresponding nodes should have all been selected, which means all the nodes corresponding to x should be selected after the query.

This discussion does not hold for the queries with an upwards axis. When an open node is selected after an upwards query, it may come from a closed node and we have no guarantee that all the corresponding open nodes are selected. Therefore, we add a postprocessing of sharing the selected nodes for the upwards axes.

Algorithm 4 in **Fig. 11** shows the procedure for querying a step with parent axis. It has almost the same flow as that of the child axis (lines 1–5), except for the last call of the ShareNodes function.

Algorithm 2 QUERY⟨child⟩($pt_{[P]}$, $InputList_{[P]}$, *test*)

Input: $pt_{[P]}$: an indexed set of partial trees
$InputList_{[P]}$: an indexed set of input nodes
test: a string of nametest
Output: an indexed set of results

```
1: for p ∈ [0, P) do
2:   OutputList_p ← []
3:   for all n ∈ InputList_p do
4:     OutputList_p
         ← OutputList_p ∪ [nc | nc ∈ n.children, nc.tag = test]
5: return OutputList_[P]
```

Algorithm 3 QUERY⟨descendant⟩($pt_{[P]}$, $InputList_{[P]}$, *test*)

Input: $pt_{[P]}$: an indexed set of partial trees
$InputList_{[P]}$: an indexed set of input nodes
test: a string of nametest
Output: an indexed set of results

```
1: for p ∈ [0, P) do
2:   SetIsChecked(pt_p, false)
3:   OutputList ← []
4:   for all n ∈ InputList_p do
5:     Stack ← {n}
6:     while not Stack.Empty() do
7:       nt ← Stack.Pop()
8:       if nt.isChecked then continue
9:       nt.isChecked ← TRUE
10:      OutputList_p
           ← OutputList_p ∪ [nc | nc ∈ nt.children, nc.tag = test]
11:      Stack.PushAll(nt.children)
12: return OutputList_[P]
```

Fig. 10 Query algorithm for downwards axes.

The SHAREDNODES function in Fig. 11 keeps open node consistency. It consists of two parts [*2]. First, it collects all the selected open nodes from partial trees (lines 4–6). Then, based on the range information of node n ($n.start$ and $n.end$), we add all the corresponding selected nodes to all the partial trees.

Now, we continue our running example Q1. For the `parent::B` after the `descendant::C`, we first directly select the parent nodes of the intermediate results independently. The results are as follows.

pt_0	pt_1	pt_2	pt_3	pt_4
$[B_1]$	[ ]	$[+B_7, +B_6+]$	[ ]	$[B_{17}]$

Here, unfortunately node $+B_7$ is selected on pt_2, but its corresponding node on pt_1 is not selected.

We then compute the SHAREDNODES function. By collecting all the open nodes from all the partial trees, we have a list $[+B_7, +B_6+]$. Since they have ranges $(1, 2)$ and $(1, 3)$, respectively, pt_1 receives two nodes B_7+ and B_6+, pt_2 receives two nodes

[*2] In our implementation of this SHAREDNODES function, there are two phases of communication: all the partial trees send their open nodes to a process and then the necessary data for a partial tree are sent back.

Algorithm 4 QUERY⟨parent⟩($pt_{[P]}$, $InputList_{[P]}$, *test*)

Input: $pt_{[P]}$: an indexed set of partial trees
$InputList_{[P]}$: an indexed set of input nodes
test: a string of nametest
Output: an indexed set of results

```
1: for p ∈ [0, P) do
2:   OutputList_p ← [ ]
3:   for all n ∈ InputList_p do
4:     if n.parent ≠ NULL and n.parent.tag = test then
5:       OutputList_p.Add(n)
6: return SHAREDNODES(pt_[P], OutputList_[P])
```

Algorithms 5 SHAREDNODES($pt_{[P]}$, $NodeList_{[P]}$)

Input: $pt_{[P]}$: an indexed set of partial trees
$NodeList_{[P]}$: an indexed set of nodes
Output: an indexed set of nodes after sharing

```
1: /* Select all open nodes and append them to a node list */
2: ToBeShared ← [ ]
3: for p ∈ [0, P) do
4:   OpenNodes ← [n | n ∈ NodeList_p,
5:                  n.type ∈ {LEFTOPEN, RIGHTOPEN, PRENODE}]
6:   ToBeShared ← ToBeShared ∪ OpenNodes
7: /* Regroup nodes by partial tree id and add them to NodeList */
8: for p ∈ [0, P) do
9:   ToBeAdded_p ← [n | n ∈ ToBeShared, n.start ≤ p ≤ n.end]
10:  OutputList_p ← NodeList_p ∪ ToBeAdded_p
11: return OutputList_[P]
```

Fig. 11 Query algorithms for upwards axes.

$+B_7$ and $+B_6+$, and pt_3 receives one node $+B_6$. By taking the union with the previous intermediate results, we obtain the following final results.

pt_0	pt_1	pt_2	pt_3	pt_4
$[B_1]$	$[B_6+, B_7+]$	$[+B_7, +B_6+]$	$[+B_6]$	$[B_{17}]$

5.2.3 Intra-sibling Axes

The following- or preceding-sibling axes retrieve nodes from a set of nodes that are siblings of the current node. In our partial trees, a set of those sibling nodes might be divided into two or more partial trees. Therefore, these intra-sibling axes require querying on other partial trees in addition to the local querying.

Without loss of generality, we discuss the following-sibling axis only. Algorithm 6 in **Fig. 12** shows the procedure for querying a step of following-sibling axis, which consists of four stages: local query, preparation, regrouping, and remote query.

In the local query, we utilize the *folsib* pointer and the *isChecked* flag to realize linear-time querying (lines 6–10). Then, in the preparation, we select the nodes that are passed to another partial tree to perform the remote query. The latter two conditions (lines 14, 15) are rather easy: we will ask a remote query if the parent node can have more segments on the right (i.e., right open). The former condition (line 13) is a little tricky. Even if the latter two conditions hold, we do not need a remote query if the node it-

self is right open. Notice that if a node is right open then it should have a corresponding left-open node on another partial tree, and that node will ask for a remote query. The regrouping is almost the same as that in ShareNodes, and the difference is the range we consider (we only look at the right for the following-sibling). Finally, the remote query finds the children from the intermediate nodes given by regrouping.

Now, let us look at our running example Q2. After `descendant::B`, we have the following results.

pt_0	pt_1	pt_2	pt_3	pt_4
$[B_1]$	$[B_6+, B_7+]$	$[+B_6+, +B_7]$	$[+B_6]$	$[B_{17}, B_{20}]$

With the first phase of `following-sibling::B`, we get the following results.

pt_0	pt_1	pt_2	pt_3	pt_4
[ ]	[ ]	[ ]	[ ]	$[B_{20}]$

Then, we collect the parent nodes that satisfy the conditions (lines 13–15). Such nodes and their ranges are: A_0+ with range [1,4] (on pt_0), $+B_6+$ with range [3,3] (on pt_2), and $+A_0+$ with range [4,4] (on pt_3). By regrouping the nodes based on the partial tree id, the input nodes for the remote query are as follows.

pt_0	pt_1	pt_2	pt_3	pt_4
[ ]	$[+A_0+]$	$[+A_0+]$	$[+A_0+, +B_6]$	$[+A_0]$

Starting from these intermediate results, we query their children and obtain the following results. Note that these results are also the final results for the query since the result of a local query is a subset of this remote query.

pt_0	pt_1	pt_2	pt_3	pt_4
[ ]	$[B_6+]$	$[+B_6+]$	$[+B_6]$	$[B_{17}, B_{20}]$

5.3 Queries with Predicate

Predicates in this paper are filters that check existence of matched nodes by given steps (simple steps without predicates). Our algorithm for handling predicates consists of three 3 phase: preparing, evaluating steps in predicates, and processing predicates. The main differences of processing predicates are the elements of their intermediate data. In the evaluation of steps, we select nodes as we do for steps without predicates. In the querying in predicates, we also attach a link to the original nodes from which the predicates are evaluated. Since the upwards or intra-sibling axes may select a node on a different partial tree, the link is a pair of partial tree id and the index of nodes in the partial tree. The intermediate data will be denoted as $(x, (i, y))$ in the pseudo code or as $x \rightarrow \{pt_i.y\}$ in the running example, both of which mean node x is selected and it has a link to node y on pt_i.

5.3.1 Preparing Predicate

Algorithm 7 in **Fig. 13** shows the procedure for initializing the process of a predicate. It just copies the nodes from the input with a link to the node itself.

For example in Q3, we have the following matched nodes up to the step `descendant::B` before the predicate evaluation.

Algorithm 6 Query⟨`following-sibling`⟩($pt_{[P]}$, $InputList_{[P]}$, *test*)

Input: $pt_{[P]}$: an indexed set of partial trees
$InputList_{[P]}$: an indexed set of input nodes
test: a string of nametest
Output: an indexed set of results

```
1:  for p ∈ [0, P) do
2:    /* Local query */
3:    SetIsChecked(pt_p, false)
4:    OutputList_p ← [ ]
5:    for all n ∈ InputList_p do
6:      while n.isChecked = FALSE and n.folsib ≠ NULL do
7:        n.isChecked ← TRUE
8:        n ← n.folsib
9:        if n.tag = test then
10:         OutputList_p.Add(n)
11:   /* Preparing remote query */
12:   for all n ∈ InputList_p do
13:     if n.type ∉ {RightOpen, PreNode}
14:       and n.parent ≠ NULL
15:       and n.parent.type ∈ {RightOpen, PreNode} then
16:       ToBeQueried.Add((n.parent, p + 1, n.parent.end))
17: /* Regroup nodes by partial tree id */
18:   for p ∈ [0, P) do
19:     RemoteInput_p ← [n | (n, st, ed) ∈ ToBeQueried, st ≤ p ≤ ed]
20: /* Remote query */
21: RemoteOutput_[P] ← Query⟨child⟩(pt_[P], RemoteInput_[P], test)
22: for p ∈ [0, P) do
23:   OutputList_p ← OutputList_p ∪ RemoteOutput_p
24: return OutputList_[P]
```

Fig. 12 Algorithm for Following-sibling axis.

pt_0	pt_1	pt_2	pt_3	pt_4
$[B_1]$	$[B_6+, B_7+]$	$[+B_6+, +B_7]$	$[+B_6]$	$[B_{17}, B_{20}]$

After the call of PreparePredicate, we have the following intermediate results. Note that all the links point to the nodes themselves at the beginning.

pt_0	pt_1
$[B_1 \rightarrow \{pt_0.B_1\}]$	$[B_6+ \rightarrow \{pt_1.B_6+\}, B_7+ \rightarrow \{pt_1.B_7+\}]$

pt_2	pt_3
$[+B_6+ \rightarrow \{pt_2.+B_6+\}, +B_7 \rightarrow \{pt_2.+B_7\}]$	$[+B_6 \rightarrow \{pt_3.+B_6\}]$

pt_4
$[B_{17} \rightarrow \{pt_4.B_{17}\}, B_{20} \rightarrow \{pt_4.B_{20}\}]$

5.3.2 Evaluation of Steps in Predicate

The evaluation of steps is almost the same as that without predicate. For example, Algorithm 9 in **Fig. 14** shows the procedure for querying a step with a child axis in the predicate; the difference is the type of intermediate values and the copying of links.

There is another important difference for the descendant, an-

cestor, following-sibling, and preceding-sibling. In the querying without predicate, we used the *isChecked* flag to avoid traversing the same node more than once. In the querying in predicates, however, the different nodes may have different links and this prevents us from using the flag. As we can see in the discussion on complexity later, this modification makes the algorithm over linear.

Now we continue our running example Q3. We then apply the query `following-sibling::B` in two phases: the local query and the remote query. The local query is the same as that of the previous section. The results are as follows.

pt_0	pt_1	pt_2	pt_3	pt_4
[ ]	[ ]	[ ]	[ ]	$[B_{20} \rightarrow \{pt_4.B_{17}\}]$

The bigger difference exists in the remote queries.

pt_0	pt_1	pt_2	pt_3
[ ]	$[B_6+ \rightarrow \{pt_0.B_1\}]$	$[+B_6+ \rightarrow \{pt_0.B_1\}]$	$[+B_6 \rightarrow \{pt_0.B_1\}]$

pt_4
$[B_{17} \rightarrow \{pt_0.B_1, pt_3.+B_6\}, B_{20} \rightarrow \{pt_0.B_1, pt_3.+B_6\}]$

Although selected nodes are the same as before, they may have multiple links: selected B_{17} and B_{20} both have two links. By merging results from local and remote queries, we finally have the following intermediate results after `following-sibling::B` in the predicate.

pt_0	pt_1	pt_2	pt_3
[ ]	$[B_6+ \rightarrow \{pt_0.B_1\}]$	$[+B_6+ \rightarrow \{pt_0.B_1\}]$	$[+B_6 \rightarrow \{pt_0.B_1\}]$

pt_4
$[B_{17} \rightarrow \{pt_0.B_1, pt_3.+B_6\}, B_{20} \rightarrow \{pt_0.B_1, pt_3.+B_6, pt_4.B_{17}\}]$

Similarly, by applying the following step `child::C`, the intermediate results are as follows.

pt_0	pt_1	pt_2	pt_3	pt_4
[ ]	[ ]	$[+C_{11} \rightarrow \{pt_0.B_1\}]$	[ ]	$[C_{19} \rightarrow \{pt_0.B_1, pt_3.+B_6\}]$

5.3.3 Processing Predicate

Finally, we process the intermediate results to obtain the results after filtering. Algorithm 8 in Fig. 13 shows the procedure for processing the predicate.

The algorithm is similar to the ShareNodes function, but in this case we consider all the results instead of open nodes. First, we collect all the links (lines 3–4) and then select only the nodes that have at least one link to the node (lines 5–6). Since there is no guarantee that all the corresponding open nodes have been activated by predicates, we need an additional call of ShareNodes.

For our running example Q3, Algorithm 8 works as follows. Links $C_{11} \rightarrow \{pt_0.B_1\}$ in the intermediate results of pt_2 adds node B_1 to the result list of pt_0 and $C_{19} \rightarrow \{pt_0.B_1, pt_3.+B_6\}$ in the intermediate results of pt_4 adds two nodes, B_1 on pt_0 and $+B_6$ on pt_3, respectively. We then apply the ShareNodes function and obtain the following intermediate results.

pt_0	pt_1	pt_2	pt_3	pt_4
$[B_1]$	$[B_6+]$	$[+B_6+]$	$[+B_6]$	[ ]

Algorithm 7 PreparePredicate($InputList_{[P]}$)

Input: $InputList_{[P]}$: an indexed set of lists of nodes

Output: an indexed set of lists of (node, link)

1: **for** $i \in [0, P)$ **do**
2: $\quad OutputList_p \leftarrow [(n, (p, n.uid)) | n \in InputList_p]$
3: **return** *OutputList*

Algorithm 8 ProcessPredicate($pt_{[P]}$, $InputList_{[P]}$)

Input: $pt_{[P]}$: an indexed set of partial trees
$\quad InputList_{[P]}$: an indexed set of lists of (node, link)

Output: an indexed set of lists of filtered nodes

1: /* regroup links by partial tree id. */
2: $AllLinks \leftarrow [\,]$
3: **for** $i \in [0, P)$ **do**
4: $\quad AllLinks \leftarrow AllLinks \cup [(p', i') | (n', (p', i')) \in InputList_p]$
5: **for** $i \in [0, P)$ **do**
6: $\quad Activated_p \leftarrow [n \mid (p', i') \in AllLinks, p = p', n.uid = i']$
7: **return** ShareNodes($pt_{[P]}$, $Activated_{[P]}$)

Fig. 13 Query algorithm for handling predicate.

Algorithm 9 PQuery⟨child⟩($pt_{[P]}$, $InputList_{[P]}$, $test_{[P]}$)

Input: $pt_{[P]}$: an indexed set of partial trees
$\quad InputList_{[P]}$: an indexed set of lists of (node, link)
$\quad test$: a string of nametest

Output: an indexed set of lists of (node, link)

1: **for** $p \in [0, P)$ **do**
2: $\quad OutputList_p \leftarrow []$
3: $\quad$ **for all** $(n, link) \in InputList_p$ **do**
4: $\quad\quad OutputList_p$
$\quad\quad\quad \leftarrow OutputList_p \cup [(nc, link) \mid nc \in n.children, nc.tag = test]$
5: **return** $OutputList_{[P]}$

Fig. 14 Query algorithm for child axis in a predicate.

The last step is simply calling the processing of the step with child axis, and the final results for Q3 are as follows.

pt_0	pt_1	pt_2	pt_3	pt_4
$[C_2]$	[ ]	$[C_{11}]$	[ ]	[ ]

Then, the query of Q3 is completed. All the nodes in the result lists are the final results.

5.4 Worst-case Complexity

At the end of this section, we discuss the time complexity of our algorithms. Here we analyze the worst-case complexity in the following categorization:

- axes,
- without or in predicate, and
- local computation and network communication.

For discussion, let N be the total number of nodes in a given XML document, H be the tree height, and P be the number of partial trees. Assuming that the given document is evenly split, the number of nodes in a chunk is N/P. Each partial tree may have pre-path, which has at most H extra nodes. Therefore, the

Table 5 Time complexity.

	without predicate		in predicate	
	computation	network	computation	network
child	$O(N/P+H)$	0	$O(N/P+PK^2)$	0
descendant	$O(N/P+H)$	0	$O(KN/P+PK^2)$	0
parent	$O(K)$	$O(PH)$	$O(PK^2)$	$O(P^2HK)$
ancestor	$O(N/P+H)$	$O(PH)$	$O(KN/P+PK^2)$	$O(P^2HK)$
folsib	$O(N/P+H)$	$O(PH)$	$O(KN/P+PK^2)$	$O(P^2HK)$
prepare			$O(N/P+H)$	0
process			$O(P^2K^2)$	$O(P^2K^2)$

number of nodes in a partial tree is at most $N/P+H$. The number of open nodes are at most $2H$. Let the number of nodes in the intermediate results be K; this is also the size of the input for processing a step.

Table 5 shows the time complexity of the axes without or with predicates. We discuss some important points with regard to the time complexity.

For the querying without predicate, the local computation cost is linear with respect to the size of the tree. Naive implementation of the descendant, ancestor, or following-sibling would have squared the cost. In our algorithm, we obtained the linear cost by using the *isChecked* flag.

For the downwards axes (child and descendant) and to prepare predicates, we need no communication. For the parent, ancestor, and following-sibling, we require communication. The amount of data to be exchanged is $O(PH)$. With these results, the total complexity of our XPath query algorithm is $O(N/P+PH)$ if we have no predicates. This is a cost optimal algorithm under $P < \sqrt{N/H}$.

When there are predicates, the worst-case complexity becomes much worse. The two main reasons are as following.

- Due to the links, we cannot simply use the *isChecked* flag. This introduces additional factor K for the computation.
- The number of links is at most PK for each node. If all the open or matched nodes on all the partial trees have that many links, then the total amount of network transfer becomes $O(P^2HK)$ or $O(P^2K^2)$.

By summing all the terms, the time complexity of querying XPath with predicate is bound by $O(KN/P+P^2K^2)$.

6. Experiments and Discussion

In this section, we report the results of the experiments conducted to test the efficiency of our algorithms. The experiments test two queries on difference sizes of machine-generated XML documents, ranging from 669 MB to 8 GB.

6.1 Experimental Data

The experimental data we used in our experiments are generated by xmlgen, which is an XML document generator developed under the an XML benchmark project, XMark [24]. The XMark project aims to provide a benchmark suite that allows users and developers to gain insights into the characteristics of their XML repositories. It produces XML documents modeling an auction website, a typical e-commerce application. The xmlgen-generated data are well-formed, valid, and meaningful to the size of several GBytes. The number and type of elements are chosen in accordance with a template and parameterized with certain probability distributions. The characteristics of the document are fully preserved under scaling, aiding the analysis of bottlenecks and how they evolve with increasing data volume.

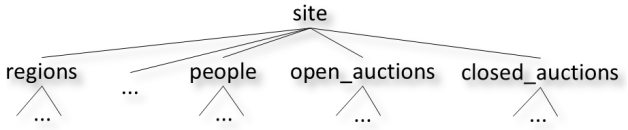

Fig. 15 Structure of xmlgen generated XML tree.

The xmlgen generates XML documents by repeating nodes many times from a model tree, which means we can obtain the tree with the same structure when nodes that have the same tag names on the same level are removed. This root and first level of this model are shown in **Fig. 15**. The root `site` has many nodes with different names (actually, there are 7 children). The xmlgen generates XML documents of different sizes with only one input parameter. For example, we can create 1 GB of file by a parameter of 9. In our experiments, the sizes of XML files created by xmlgen range from 669 MB (the parameter is 7) to 8 GB (the parameter is 72).

To show the effectiveness of our parallel XPath query algorithm, we perform two types of tests. In the first test, we applied queries to an XML file of 669 MB to show the scalability of the algorithm with respect to the number of PCs. In the second test, we fixed the number of PCs to 16 and increased the size of the XML documents.

We use the queries in **Table 6** for our tests. The first three queries Q4, Q5, and Q6 test the scalability and data processing ability. The last Q7, which has the most steps, is to test how much the network communications affect the performance.

6.2 Hardware

The algorithm was implemented under a server/client architecture programmed in Java. The server ran on a single PC, which had an Intel(R) Core(TM) i5-760 CPU @ 2.80 GHz CPU, 8 GB of memory, and the OS and Java environment were Windows 7 and Java 1.8. The clients ran on at most 16 PCs in a PC cluster, where 9 PCs had Intel(R) Core(TM) i5-2500 @ 3.30 GHz CPU, 7 PCs had Intel(R) have Core(TM) i5 CPU 760 @ 2.80 GHz, 8 GB of memory, and the OS and Java environment were Ubuntu 14.04 LTS (Linux kernel: 3.16.0-41-generic) and Java 1.8. We solely focus on the performance of our algorithms querying on partial trees; thus, we construct only one partial tree for each client computer and do not use any multi-thread techniques, hyper threading, or other memory-sharing techniques.

6.3 Parallel Speedups

This first experiment tested the efficiency and parallel speedups of querying with respect to the number of PCs. We use an XML file of size 669 MB and the three queries Q4, Q5, and Q6 with 1, 2, 4, 8 and 16 client PCs. The size of XML data on one single computer we have tested is 669 MB. We split the input trees evenly by size.

Figure 16 shows the speedups and time taken for the three

Table 6 Queries used for the experiments.

Q4	/child::site/descendant::keyword/parent::text
Q5	/child::site/child::people/child::person[child::profile/child::gender]/child::name
Q6	/child::site/child::open_auctions/child::open_auction/child::bidder[following-sibling::bidder]
Q7	/child::site/child::closed_auctions/child::closed_auction/child::annotation/child::description/child::text/child::keyword

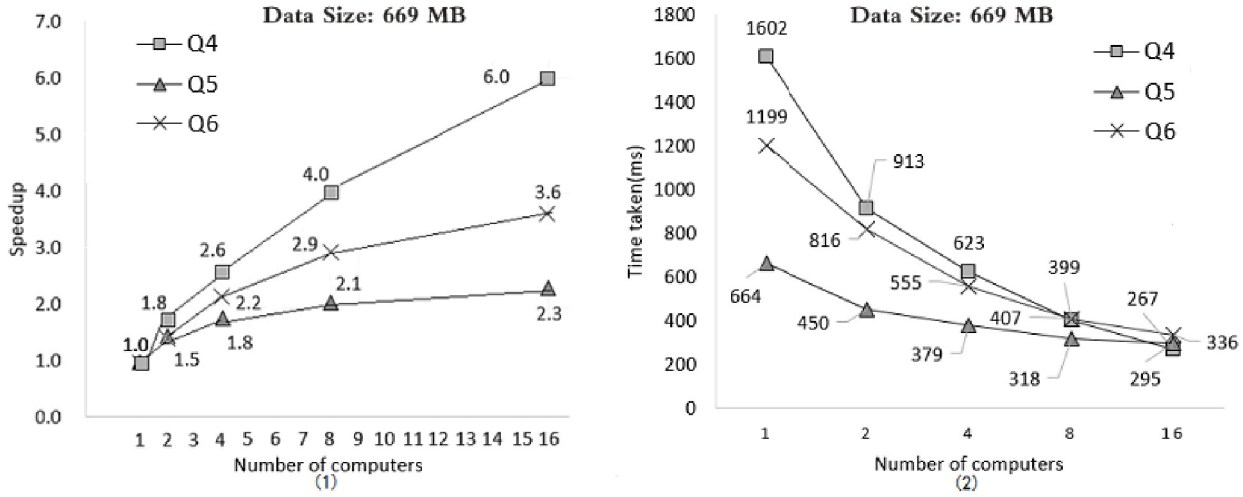

Fig. 16 Speedups and time with respect to the number of clients PCs.

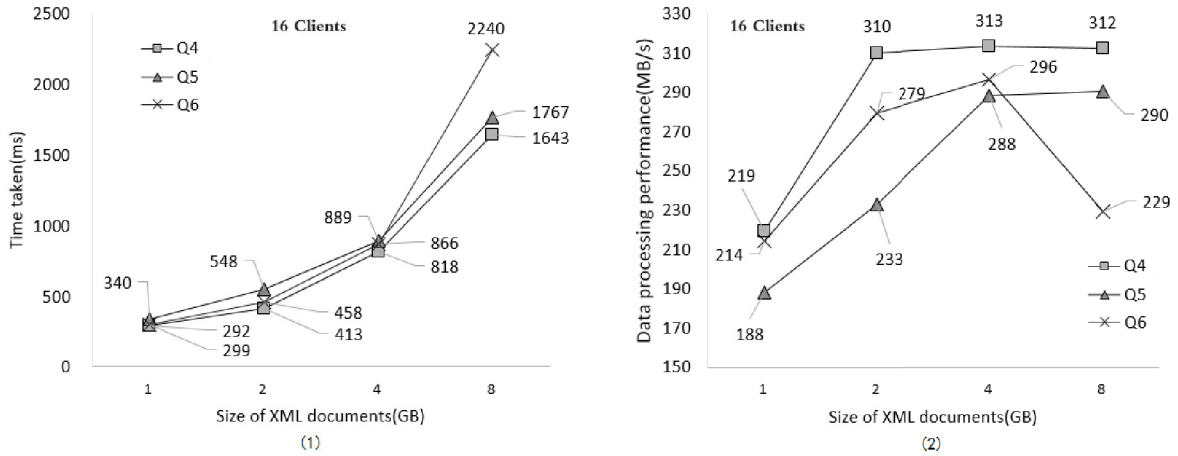

Fig. 17 Time taken and data processing ability.

queries. The time taken is significantly reduced for all three queries as the number of clients increases. It is more apparent for Q4. Because Q4 has no predicate and the communication affects the overall time less. We achieved speedups of a factor of 6.0 for Q4 with 16 clients. Both Q5 and Q6 have a predicate, and it requires more communication phases. The speedups for them are relatively low, a factor of 3.6 for Q6 and 2.3 for Q5. We will discuss these lower speedups by analyzing the network communication cost in Section 6.5.

6.4 Scalability

The second experiment is designed to evaluate the performance of data processing ability per computer as the sizes of input XML data increase. The size of XML data on a single computer that can be processed is limited to around 669 MB because of the limit of the size of memory of a single computer. Some necessary fields and variables of a single node are declared, which takes extra space that is more than the size of the original XML string. We split input data at a maximum of 512 MB for a single computer. We use 16 computes for computation, and the sizes of the input XML data range from 1 GB to 8 GB. The time taken is shown in **Fig. 17** (1).

The times taken of Q4 and Q5 are almost doubled as the sizes of the input XML data doubled as shown in Fig. 17 (1). For Q6, the time taken is doubled when the size of data increases from 1 GB to 2 GB and 2 GB to 4 GB. However, when the size of data increases from 4 GB to 8 GB, the time taken is a little more than doubled. We believe that the extra time is likely caused by the cost by the rapid increase of intermediate data and the cost of network communication. We will discuss the cost of the network communication in the following section.

From Fig. 17 (2), the data processing ability of a single com-

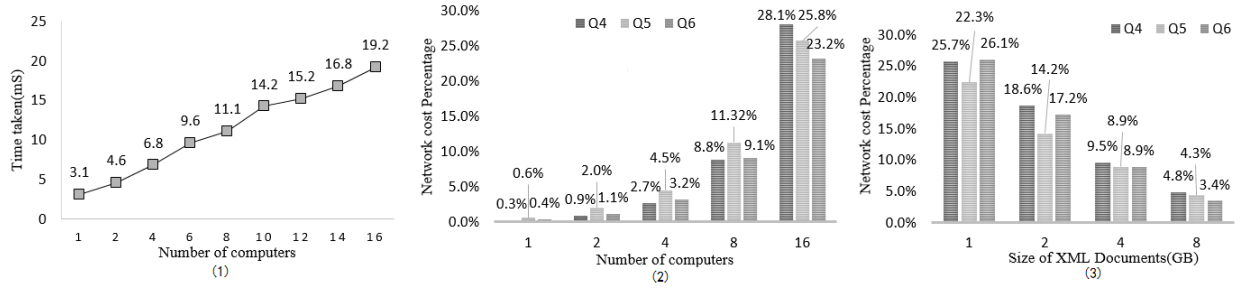

Fig. 18 Cost of network communication.

puter ranges from about 200 MB/s to 300 MB/s. We also find that when the size of data increases, the data processing ability increases as well. It also attributes to the cost of the network communication.

6.5 Communication Cost

Our algorithms are implemented with server/client architecture. The communication between the server and the clients is based on TCP/IP protocol. The server is in charge of partial tree construction and evaluation control of queries. It holds the information of open nodes of all the partial trees, including the ranges. The communication mechanism in our framework is based on string messages. We tested the communication between the server and the clients. The results are shown in **Fig. 18**.

As we can see in Fig. 18 (1), when a new client is added, it takes approximately 1 extra millisecond for a single step on the network. For 16 clients, each query step takes around 20 ms for network communication. We also tested Q7. Even if we use more computers, we obtain a speedup of less than 1, which means the efficiency is slowed down by the cost of network communication.

For Q7, the efficiency goes down even when the number of clients increases. The reason is that for each step of the Q7, it takes around 20 ms for the communication when 16 clients are used. In addition, the child axis does not take too much time for evaluation. Therefore, the time saved by increasing the number of clients is wasted due to the cost of network communication.

We also evaluated the cost of network communication for the former two experiments. For the experiments in Section 6.3, as we can see in Fig. 18 (2), the more computers are used, the more time is taken on network communication. That is why the speedup increase was not obvious when more computers were added. We can obtain the same conclusion from Fig. 18 (3).

For the experiments in this section, since the number of computers are the same, the time taken on network is almost the same. When the size of the data increases, the effect of network communication becomes less. Therefore, we can conclude that we can obtain better performance as the size of the data increases. For the network, if we could improve the implementation to reduce the cost of network communication, we could get better performance for a small amount of data.

6.6 Imbalance of Xmlgen-generated XML Tree

After analyzing these data carefully, we find that our algorithm does not show the best performance by using these XML documents, because of the imbalanced structure of xmlgen-generated XML documents.

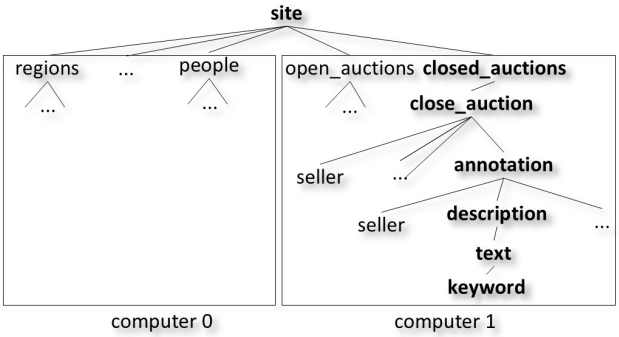

Fig. 19 An example of tree assignment to computers.

In Fig. 15, the children of the root have different tags. When we evaluate an XPath query on this tree by using our algorithm, we sometimes only query a small part of the tree, which means we apply queries on some of the partial trees that contain specific nodes while we do nothing to the other.

For example, as we can see in **Fig. 19**, the highlighted parts are the traversed edges of the tree by Q7. We use two computers for the query. The specific nodes are all on the first partial tree on computer 1. While on computer 0, there is no node that has the tag name "`close_auction`" as child of `site`; therefore, computer 1 is idle after testing the second step of Q7.

From our experiments, we also find that only part of the computers take part in the computation. When we use only one computer for loading 669 MB of data, the total number of nodes is 10023967. And there are 15300 `person` nodes for Q5 and 72000 `closed_auction` nodes for Q6. As we can see in **Table 7** and **Table 8**, we only obtain 2 out of PCs 4 and 3 out of 16 PCs that have a result node after processing `/child::person` in Q5. That means only a few of the PCs are used in the query while the others are idle. We also notice that when the number of computers increased from 4 to 16, the number of the computers for computation only increased from 2 to 3 for `child::people`. Thus, we cannot utilize the total number of PCs. We can obtain the same conclusion when we take a look at the data for processing `/child::close_auction` step in Q6.

One more thing we have noticed is the imbalance of nodes distribution of partial trees. Note that for PC_9 or PC_{10}, there are almost three times more nodes compared to others. Due to the fact that we need to wait until all the computers' work is done for the next step, the imbalanced distribution of nodes also reduces the speedup of these experiments.

Table 7 16 PCs are used for steps in Q5 and Q6.

PC id	Nodes Count	/person	/open_auctions
1	442240	0	0
2	444828	0	0
3	442158	0	0
4	444515	0	0
5	441671	0	0
6	442697	0	0
7	442021	0	0
8	546640	14247	0
9	1331617	102551	0
10	1042920	36204	12032
11	886279	0	18666
12	890306	0	18643
13	891300	0	18720
14	591080	0	3943
15	513317	0	0
16	230460	0	0

Table 8 4PCs are used for steps in Q5 and Q6.

PC id	Nodes Count	/person	/open_auctions
1	1773703	0	0
2	1872954	14242	0
3	4151158	138759	49338
4	2226168	0	22663

From the above discussion, we come to the conclusion that the structure of input XML documents affects the speedups of the experiments. If we apply our algorithm to some well-balanced XML trees, we could obtain better speedups related to the increase in the number of computers.

7. Related Work

Many papers have addressed the topic of implementations of XPath queries in parallel. One significant paper was presented by IBM [3]. The paper proposes three kinds of strategies for XPath queries in parallel: data partition strategy, query partition strategy, and hybrid partition strategy. Many papers can be categorized to one of the strategies. References [11], [15], [25] focus on XPath queries implemented in a shared-memory environment. References [19], [23] focused on XML parsing, which is related to our parsing algorithm. Reference [1] proposed ideas about XML processing techniques that are helpful for our research. Some prior researches are based on a common assumption that a large amount of XPath queries are executed over an XML stream. YFilter [5] and XMLTK [2] execute thousands of small queries in parallel. The parsing phase is still sequential. Indexing is also a hot topic for improving the performance of parallel XML queries processing. References [4], [10], [12] are related to this field. They examined the indices on different types of trees, including B+-tree, R-tree, and XR-tree.

The idea of dividing the XML documents and running the computation for trees with the chunks is not new. Kakehi et al. [13] showed a parallel tree reduction algorithm from the nodes in chunks. Based on the idea given by Kakehi et al., Emoto and Imachi [6] developed a parallel tree reduction algorithm on Hadoop, and Matsuzaki and Miyazaki [17] developed a parallel tree accumulation algorithm. A similar approach was taken by Sevilgen et al. [20] who developed a simpler version of tree accumulations over the serialized representation of trees.

It is known that we can develop a parallel algorithm for XPath queries using the tree accumulations [16]. The approach we took in this paper is inspired by the work by Morihata [18]. To discuss the advantages of the proposed algorithm and compare by implementation with other approaches are our important future work.

8. Conclusion

In this paper, we proposed algorithms for a subset of XPath queries on a large XML tree in parallel and implemented them on a 16-node PC cluster. We developed our own framework for the experiments. The experiment results showed a speedup of a factor of 6 on a 16-node PC cluster.

Acknowledgments Part of this work was supported by JSPS KAKENHI No.25330088.

References

[1] Andriescu, E.-M., Azzabi, A. and Hains, G.: Parallel processing of Forward XPath queries: An experiment with BSML, *TR-LACL*, Vol.11 (2010).
[2] Avila-Campillo, I., Green, T.J., et al.: XMLTK: An XML toolkit for scalable XML stream processing, *Technical report, PlanX* (2002).
[3] Bordawekar, R., Lim, L., Kementsietsidis, A. and Kok, B.: To Parallelize or Not to Parallelize: XPath Queries on Multi-core Systems, *IBM Research Report* (2009).
[4] Chien, S.Y., Vagena, Z., Zhang, D., Tsotras, V.J. and Zaniolo, C.: Efficient Structural Joins on Indexed XML Documents, *VLDB 2002*, pp.263–274 (2002).
[5] Diao, Y., Fischer, P., Franklin, M., et al.: YFilter: Efficient and scalable filt. of XML doc. In ICDE, pp.341–342 (2002).
[6] Emoto, K. and Imachi, H.: Parallel tree reduction on MapReduce, *Proc. International Conference on Computational Science* (*ICCS 2012*), *Procedia Computer Science*, Vol.9, pp.1827–1836, Elsevier (2012).
[7] Franceschet, M. and XPathMark: Functional and performance tests for XPath, *XQuery Implementation Paradigms*, *Dagstuhl Seminar Proceedings*, No.06472, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007).
[8] Francis, N., David, C. and Libkin, L.: A direct transation from XPath to nondet. automata, *Workshop on Foundations of Data Management*, pp.350–361 (2011).
[9] Gazit, H., Miller, G.L. and Teng, S.-H.: Optimal tree contraction in EREW model, *Proc. Princeton Workshop on Algorithms, Architectures, and Technical Issues for Models of Concurrent Computation, 1987 APLAS*, pp.139–156 (2002).
[10] Grust, T.: Accelerating XPath Location Steps, *SIGMOD 2002*, pp.109–120 (2002).
[11] Huang, X., Si, X., Yuan, X. and Wang, C.: A Dynamic Load-balancing Scheme for XPath Queries Parallelization in Shared Memory Multicore Systems, *Journal of Computers*, pp.1436–1445 (2014).
[12] Jiang, H.F., Lu, H.J., Chin, B. and Wang, W.: XR-Tree: Indexing XML Data for Efficient Structural Joins, *ICDE*, pp.253–264 (2003).
[13] Kakehi, K., Matsuzaki, K. and Emoto, K.: Efficient Parallel Tree Reductions on Distributed Memory Environments, *7th International Conference on Computational Science* (*ICCS2007*), pp.601–608 (2007).
[14] Kawamura, K. and Matsuzaki, K.: Dividing Huge XML Trees Using the m-bridge Technique over One-to-one orresponding Binary Trees, *IPSJ Trans. Programming*, pp.40–50 (2014).
[15] Krulis, E. and Yaghob, E.: Efficient Implementation of XPath Processor on Multi-Core CPUs, pp.60–71 (2010).
[16] Matsuzaki, K.: Parallel Programming with Tree Skeleton, Ph.D. The-

sis, The University of Tokyo (2007).
[17] Matsuzaki, K. and Miyazaki, R.: Parallel Tree Accumulations on MapReduce, *International Journal of Parallel Programming*, online (2015).
[18] Morihata, A.: Work Efficient Distributed-Memory Parallel Evaluation of XPath Queries, *30th JSSST Conference Proceedings* (2013) (in Japanese).
[19] Pan, Y., Lu, W., Zhang, Y. and Chiu, K.: A static load-balancing scheme for parallel XML parsing on multicore CPUs, *IEEE International Symposium on Cluster Computing and the Grid*, Janeiro, pp.351–362 (2007).
[20] Sevilgen, F.E., Aluru, S. and Futamura, N.: Parallel Algorithms for Tree Accumulations, *Journal of Parallel and Distributed Computing*, Vol.65, No.1, pp.85–93 (2005).
[21] W3C, XML Tutorial, available from ⟨http://www.w3schools.com/xml/⟩ (accessed 2010).
[22] W3C, XML Path Language (XPath) 2.0 (Second Edition), available from ⟨http://www.w3.org/TR/xpath20/⟩ (accessed 2010).
[23] Wu, Y., Zhang, Q., Yu, Z. and Li, J.: A Hybrid Parallel Processing for XML Parsing and Schema Validation, *The Markup Conference, Cluster Computing and the Grid, Seventh IEEE International Symposium*, pp.351–362 (2008).
[24] XMark, XMark - An XML Benchmark Project, available from ⟨http://www.xml-benchmark.org/generator.html⟩ (accessed 2010).
[25] Zhang, Y., Pan, Y. and Chiu, K.: A Parallel XPath Engine Based on Concurrent NFA Execution, *IEEE 16th International Conference on Parallel and Distributed Systems*, pp.314–321 (2010).

Wei Hao is a Ph.D. student in Kochi University of Technology in Japan and is a Lecturer in Anhui University of Science and Technology in China. He received his B.E. and M.S. from Anhui University of Science and Technology in 2004 and 2007, respectively, and began doctoral study since 2014. His current research interest is in parallel programming and XML data processing.

Kiminori Matsuzaki is an Associate Professor of Kochi University of Technology in Japan. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2001, 2003 and 2007, respectively. He was an Assistant Professor (2005–2009) in The University of Tokyo, before joining Kochi University of Technology as an Associate Professor in 2009. His research interest is in parallel programming and algorithm derivation. He is a member of ACM, JSSST, IEEE.