Parallelization of XPath Queries on Large XML Documents

Parallelization of XPath Queries on Large XML Documents

by

Wei Hao
Student ID Number: 1188004

A dissertation submitted to the
Engineering Course, Department of Engineering,
Graduate School of Engineering,
Kochi University of Technology,
Kochi, Japan

in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Assessment Committee:

Supervisor:	Kiminori Matsuzaki, School of Information
Co-Supervisor:	Makoto Iwata, School of Information
Co-Supervisor:	Tomoharu Ugawa, School of Information
Committee Member:	Kazutoshi Yokoyama, School of Information
Committee Member:	Yukinobu Hoshino, School of Systems Engineering

September 20, 2018

Abstract

In recent decades, the volume of information increases dramaticly, leading to an urgent demand for high-performance data processing technologies. XML document processing as a common and popularly used information processing technique has been intensively studied.

About 20 years ago at the early stage of XML processing, studies mainly focused on the sequential approaches, which were limited by the fact that CPUs at the time were commonly single-core processors. In the recent decade, with the development of multiple-core CPUs, it provides not only more cores we can use in a single CPU, but also better availability with cheaper prices. Therefore, parallelization become popular in information processing.

Parallelization of XPath queries over XML documents became popular since the recent decade. At the time, studies focused on a small set of XPath queries and were designed to process XML documents in shared-memory environments. Therefore, they were not practical for processing large XML documents and it was difficult for them to meet the requirements of processing rapidly grown large XML documents.

To overcome the difficulties, we first revived an existing study proposed by Bordawekar et al. in 2008. Their work was implemented on an XSLT processor Xalan and has already been out of date now due to the developments of hardware and software. We presented our three implementations on top of a state-of-the-art XML databases engine BaseX over XML documents sized server gigabytes. Since BaseX provides full support for XPath/XQuery 3.1, we can harness this feature to process sub-queries from the division of target XPath queries.

This study establishes the availability of Bordawekar et al.'s work. Then, we propose a fragmentation approach that divides an XML document into node-balanced subtrees with randomization for achieving better load-balance. Combined with the previous data partitioning strategy, we show a promising approach for processing large XML documents efficiently in distributed-memory environments.

The previous partition and fragmentation based study enable us to easily process

large XML documents in distributed-memory environments. However, it still has its flaw that is limited to top-down queries. Therefore, to enrich the expressiveness of queries that can be processed in our study, we then proposed a novel tree, called partial tree. With partial tree, we can make the XML processing support more types of queries, making it more feasible to process large XML documents by utilizing computer clusters. We also propose an efficient indexing scheme for representing partial tree so that we can achieve better query performance.

There are two important contributions proposed in the thesis.

The first contribution involves three implementations of Bordawekar et al.'s partitioning strategies and our observations and perspectives from the experiment results. Our implementations are designed for the parallelization of XPath queries on top of BaseX. With these implementations, XPath queries can be easily parallelized by simply rewriting XPath queries with XQuery expressions. We conduct experiments to evaluate our implementations and the results showed that these implementations achieved significant speedups over two large XML documents. Besides the experiment results, we also present significant observations and perspectives from the experiment results. Then, we present a proposal to extend the fragmentation algorithms to exploit data partitioning strategy in distributed-memory environments.

The second contribution is the design of a novel tree structure, called partial tree, for parallel XML processing. With this tree structure, we can split an XML document into multiple chunks and represent each of the chunks with partial trees. We also designed a series of algorithms for evaluating queries over these partial trees. Since the partial trees are created from separated chunks, we can distribute these chunks to computer clusters. In this way, we can run queries on them in distributed-memory environments. Then, we propose an efficient implementation of partial tree by index, which is an indexing scheme, called BFS-array index along with grouped index. With this indexing scheme, we can implement partial tree efficiently, in both memory consumption and absolute query performance. The experiments showed that the implementation can process 100s GB of XML documents with 32 EC2 computers. The execution times were only seconds for most queries used in the experiments and the throughput was

approximately 1 GB/s. The experiment also showed that with partial tree we can achieve a speedup of up to 36.6 with 64 workers on 8 computers for the queries used in the experiment.

Contents

List of Figures

List of Tables

LIST OF TABLES

Acknowledgements

There were many persons who provided me a lot of assistance with this work. Without their assistance, I could not finish this thesis. Therefore, I would like to give my sincere gratitude to them, particularly the following professors, classmates, family members, friends etc.

First and foremost, I would like to give my sincerest gratitude to my doctoral supervisor Professor Kiminori Matsuzaki. It is my greatest honor to be his first Ph.D. student. Professor Kiminori Matsuzaki is a very kind and amiable person with the consistent solid support both on my Ph.D. research work and daily life in Japan. I appreciate all his contributions of energetic enthusiasm, immense knowledge, and experience on research, insightful ideas, and generous support, making my Ph.D. experience productive and fruitful. I am also thankful to his excellent advice and examples he has provided as a successful computer scientist and professor.

I would like to express sincere appreciation to Dr.Shigeyuki Sato for his great help on my research work. I have quite often been enlightened by his quite strict attitude towards research work and setting such a good example for me.

I would like to think Associate Professor Tomoharu Ugawa for his helpful advice on my research. I would like to thank Professor Jingzhao Li, who assisted me with some job issues back in China.

I would like to thank my father Wenlin Hao and my mother Qinglin Zhu for their significant supports and encouragements.

I would like to thank the following friends: Onofre Coll Ruiz, who was my classmate and lab mate. He gave me a great many help in English learning and daily life. I would like to thank Naudia Patterson, who is an English teacher and assisted me with revising my thesis.

I would like to express sincere appreciation to Kochi University of Technology for providing me such a great opportunity for doing research.

Chapter 1

Introduction

1.1 Background

XML is a popular language to represent arbitrary data in XML documents. XML processing is abut how to store, represent, query XML documents, which has been intensively studied. In recent decades, with the dramatic growth of the volume of information, there is an urgent demand for high-performance data processing technologies. This change also requires XML processing to be further studied to meet the requirements of processing large XML documents as the size of XML documents is increasing greatly. For example, Wikipedia [5] provides a dump service [6] that exports wikitext source and metadata embedded in XML documents. The size of the data were less than one gigabyte before 2006. Just 10 years later, it increased to over 100 gigabytes [4]. Some XML documents have even exceeded over a hundred of gigabyte. For example, an online database of protein sequence UniProtKB [90] can be stored in a single XML document of 358 GB.

Up to the dawn of this century, studies of XML processing mainly focused on sequential processing [12, 46, 48, 71, 78] due to the fact that multi-core CPUs were not popular at the time. As the multi-core CPUs became dominant gradually, more researches shifted to parallel XML processing and thus many related studies were proposed [67, 84, 87, 88, 98].

To obtain desired data from an XML document, XPath is popularly path language

used to denote a query to the desired data. It is thus been widely studies. To process XML document efficiently by using multi-core CPUs, one common topic in the field of parallel XML processing is the parallelization of XPath [95] queries over XML documents. The idea of parallelization is to divide an XML document into multiple pieces to process queries on the divided pieces separately in parallel with multiple cores or computers, and to merge the sub results obtained from these pieces to form the final result.

Many studies were proposed [29, 34, 74, 79, 87] regarding XML processing. Most of these studies tend to represent XML document as trees presented in shared-memory where all the involved threads have access to the XML data, so as to parallelize XML processing on these trees, which usually utilize multi-thread techniques. However, these studies discussed the parallelization of XPath queries only on the a whole tree (or a number of whole trees) and did not relate to how to divide a tree. The key problem of the parallelization is how to deal with the hierarchical (or tree-shaped) structure of an XML document, which is the intrinsic characteristic of each XML document. This characteristic brings a significant challenge of parallel XML document processing[need evidence], because the hierarchical structure is more complicated than a list or an array to be divided and processed in parallel, forcing us to deal with the connections of this structural information among the parts of the divided XML document. Thus, these above approaches are not suitable for XML processing in distributed-memory environment.

Another traditional way of XML processing is to exploit database techniques. XML processing in databases has also been widely studied [43, 50, 52, 76, 82]. Common database techniques, such as indexing [57, 85, 91], join algorithms [47, 65, 66], are also valid to be applied to XML processing. Nowadays, concurrent transactions are available in modern database engines, which makes it possible and available to evaluate XPath queries on XML documents in parallel. We also consider to exploit this feature of database engines to extend the parallelization from shared-memory environment to distributed-memory environments. However, to the best of our knowledge, there is still no existing work that studies the parallelization of XPath queries based on XML

databases in a distributed-memory environment. Therefore, it is not clear how to utilize the power of XML databases in evaluating of XPath queries over large XML documents in distributed-memory environments.

In this study, we address the following two challenges for processing large XML documents in distributed-memory environments.

- Parallelizing Evaluation of XPath Query using XML databases.
  By dividing or rewriting an XPath query into multiple sub-queries, such as [20, 21], we can convert the evaluation of the original query to the evaluation of these sub-queries. However, there is a technical difficulty as to figure out how to parallelize the evaluation of a single query by exploiting existing XML database engines, because when we evaluate queries in parallel using a database engine, it often brings obvious overhead, which ruins the speedups achieved by parallel evaluation. Therefore, it is important to study how to utilize a database engine to reduce the overhead and achieve good performance gain and scalability. It is also worth studying how to distribute an XML document to multiple XML databases and process them efficiently for the same purpose.

- Generic Approach for Parallelizing Evaluation of XPath Queries.
  When processing XML documents in distributed-memory environments, dividing an XML document into chunks and distribute the processing of chunk to multiple computers is an intuitive way. However, how to represent chunks and evaluate queries on them for efficient evaluation, and how to handle the communication among computers are still challenges for efficient XML processing.

This thesis addresses the above two technical difficulties. In the following section, a brief introduction is given to illustrate the main ides for the difficulties with examples.

1.2 A Quick Walk-through

To address the parallelization of XPath queries in XML databases, there are mainly two ideas presented in this thesis, we demonstrate them with an example.

The first idea of our study involves an implementation work of [21] proposed by Bordawekar et al., which presented three different ways of exploiting existing XML processors, such as Xalan [7], to parallelize XPath quires with no need to modify the XPath processing engine. In more detail, three strategies: query partitioning, data partitioning and hybrid partitioning were proposed in their paper. Query partitioning is to split a given query into independent queries by using predicates, e.g., from q_1`[`q_2 `or` q_3`]` to q_1`[`q_2`]` and q_1`[`q_3`]`, and from q_1`/`q_2 to q_1`[position() <= n]/`q_2 and q_1`[position() > n]/`q_2. Data partitioning is to split a given query into a prefix query and a suffix query, e.g., from q_1`/`q_2 to prefix q_1 and suffix q_2, and to run the suffix query in parallel on each node of the result of the prefix query. How to merge depends on the partitioning strategies. The hybrid partitioning strategy is a mix of the first two partitioning. Although these three strategies were experimentally showed to be efficient for XML parallelization, it is worth studying how to make them work on modern XPath processors on more cores since both the hardware and software have changed a lot during the past ten years. In our study, we have developed two implementations of data-partitioning parallelization and a implementation of query-partitioning parallelization on top of BaseX [10], exploiting a state-of-the-art XML database engine, BaseX. We focus on the data partitioning and query partitioning strategies in this thesis. For query partitioning, we utilize the relationships of sub-queries in the predicate. More specifically we split a query into sub-queries by separating and-/or- predicates, and then intersect/union results to form the final resutls. For data partitioning, we split a query into a prefix query and suffix query. Then we evaluate the prefix query to obtain the intermeidate results and parallelize the evaluation of the suffix query with the intermediate results. Lastly, we concatenate results of the suffix queries in order. We also give a proposal for distributed XML processing based on our observations from the experiment results.

The second approach is based on a novel tree structure, called partial tree. A partial tree is a tree-shaped structure for representing a chunk of an XML document so that XPath queries can be evaluated on partial trees of an XML documents in parallel. Partial trees can be used in distributed-memory environments. To understand what a

partial tree is and how it works, we use the following XML document as an example (Numbers in pre-order traversal are placed along the nodes).

```
<A><B><C></C><C></C></C><C></C><A><C></C><B>
</B></A></B><A><B></B></A><B></B></A>
```

We can construct a tree as shown in Figure 1.1 for representing the given XML document. Note that each node in the tree structure is formed by parsing a pair of tags, the start tag and the end tag. The tags in between a pair of tags form nodes as children or descendants of the node formed by the pair of tags. As we know, a node in the tree structure should come from a pair of tags, but not a single tag. Thus, here comes a question: *How to parse a single tag, such as a start tag or end tag in a chunk of XML document if its other half is missing?*.

For example, let us consider the underlined part of the document, which corresponds to the nodes with color gray in Figure 1.1. The corresponding nodes of tags in the underlined part in the example are colored gray. Note that some tags, such as the first `</C>` or the last `<B>` miss their matching tags. Then, we have two questions: (1) how can we represent these tags when we parse this chunk? (2) how can we apply queries to the gray part in the figure in case we do not know the path from it to the root of the whole tree, such as A_1 and B_2 that are missing in the chunk?

To answer on the above two questions, we first proposed a novel tree structure, called partial tree. As show in Figure 1.2. We can create a partial tree for the chunk. The partial tree has the information of the missing path, i.e. A1 and B2. This tree is different from ordinary trees because we define some special nodes for partial tree (These special nodes with dots in the feature will be discussed at length in Section 5.1).

By adding the nodes on the path from the root to the current chunk part, we are now able to apply the queries to this partial tree based on the parent-child relationships. We will discuss the query algorithms in Section 5.4. Although partial tree is available in both shared-memory environments and distributed-memory environment, it is more specially designed for distributed-memory environments. This is because chunks of an XML documents can distributed to multiple computers and then be parsed into partial trees for further parallel processing.

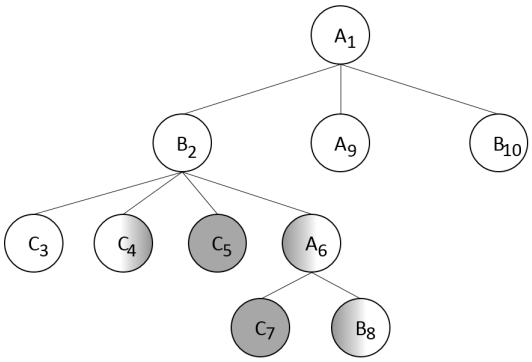


Figure 1.1: An example of XML tree with numbers in pre-order added for discussion.

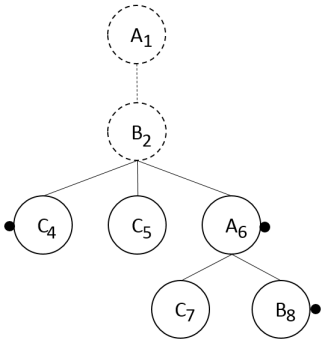


Figure 1.2: An example of partial tree (same numbering as the previous figure).

1.3 Contributions

We consider two key contributions in the thesis.

The first contribution is the approach showing how to parallelize XPath queries over fragmented XML documents stored in a number of XML databases, which also involves implementations of [21], our observations and perspectives from the experiment results. Our implementations are designed for the parallelization of XPath queries on top of BaseX, which is a state-of-the-art XML database engine and XPath/XQuery 3.1 processor. With these implementations, XPath queries can be easily parallelized by simply rewriting XPath queries with XQuery expressions. We conduct experiments to evaluate our implementations and the results showed that these implementations achieved up to 6x speedups on 12 cores for the queries over the XML documents of several gigabytes. Besides the experiment results, we also present our observations and perspectives from the experiment results. For processing larger XML documents,

we extend the study to distributed-memory environment by exploiting fragmentation that divides an input XML document into multiple fragment containing information for later querying. We then apply data partitioning on the fragments in an XML database engine BaseX.

Although we have experimentally showed that we can cheaply achieved speedups on modern XPath processors by exploiting partitioning strategies, there are many queries that we cannot process because of the evaluation on partitioned sub-queries are independant and the connection among the sub-queries are lost. To solve this problem, we proposed a novel ideas based on a tree-shaped structure. The second contribution is the design of the tree-shaped structure, called partial tree, for parallel XML processing. With this tree structure, we can split an XML document into multiple chunks and represent each of the chunks with partial trees. We also design a series of algorithms for evaluating queries over these partial trees. Since the partial trees are created from separated chunks, we can distribute these chunks to computer clusters. In this way, we can run queries on them in distributed memory environments. We also proposed an efficient implementation of partial tree. Based on some existing indexing scheme, we developed our own indexing scheme called BFS-array index along with grouped index to represent partial tree for efficient processing. With this indexing scheme, we can implement partial tree efficiently, in both memory consumption and absolute query performance. The experiments showed that the implementation can process 100s GB of XML documents with 32 EC2 computers. The execution times were only seconds for most queries used in the experiments and the throughput was approximately 1 GB/s.

1.4 Outline

In this thesis, the related work will be discussed after the quick walk-through to locate this study. Since the goal is the parallelization of XPath queries on large XML document, we will give introductions to XML and XPath. With these knowledge, we will show how to process XPath queries in parallel on a modern XPath processor BaseX by using our implementations of existing partitioning strategies. We then propose a

novel data structure for representing a part of an XML document so that we can parallelize the evaluation of XPath queries on the instances of this data structure. Lastly, draw conclusions and show the future work.

The thesis is organized in six chapters. An introduction to this study is given in Chapter 1. Here are the detailed introduction to the following Chapters.

Chapter 2

We discuss related work in three aspects: 1) XML fragmentation, which studies how to fragment an single XML document tree into multiple sub documents trees, 2) parallel evaluation of XML queries, which is about how to evaluate query on fragmented XML data, and 3) XML database techniques, which is database technology that are closely to our study.

Chapter 3

In this section, the definitions of XML and two XML query languages: XPath and XQuery used in this these are defined and we give introductions them to help understand bases of this study.

Chapter 4

We introduce our approach of [21] with an XML processing engine BaseX. we also present our implementations and evaluate our implementations on two data sets. Then, we propose our observations and perspectives from the experiment results. Based on them, we extend the study to distributed-memory environment by introducing a fragmentation approach that divides an XML document into fragments for parallel querying.

Chapter 5

Firstly, we propose a novel tree structure called partial tree and give the definitions of partial tree and related items, discuss the characteristics and design querying algorithms over partial trees. Then, we propose an efficient index scheme for partial tree. Lastly, we report and analyze the experiment results.

Chapter 6

We summarize this thesis and discuss the future work.

Chapter 2

XML and Query Languages

In this chapter, we introduce XML and two important query languages that are used in this study: XPath and XQuery.

2.1 XML

XML(eXtensible Markup Language) [2] is a standard data describing language used for representing arbitrary data in a hierarchical structure. This structure can be expressed by XML schema. An XML Schema[1] is a language for expressing constraints about XML documents at a relatively high level of abstraction.

It is also common to represent an XML document as a logical tree, in which there are many nodes of three types: *element node*, *attribute node* and *content node* as described below. These nodes are ordered according to their order of appearance in the XML document. This order is *document order*.

- Element node

  An element node is parsed from a pair of tags, a start tag and a end tag, and are used to represent the structure of an XML tree.

- Content node

  A content node (also called value node) represents the value of a element node.

[1] https://www.w3.org/standards/xml/schema

- Attribute node

  An attributes node is used to associate name-value pairs used to describe properties of the element node.

Now, we give an example to demonstrate these node types. Given the following XML text string.

```
<A>
  <B AT1="VAL1">TXT1</B>
  <D>
    <E AT2="VAL2"></E>
    <F>
      <G>
        <I></I>
      </G>
      <H>TXT2</H>
    </F>
    <J></J>
  </D>
  <K>
    <L>TXT3</L>
  </K>
</A>
```

We can construct an XML tree as shown in Figure 2.1 (The example of three element types are shown on left-bottom corner). For example, the node B is an element node. As we can see, node B has two child nodes. The left child node is an attribute node with the name 'AT1' and the value 'VAL1'. The right one is a value node with the string value 'TXT1'. As for context node, they can be found at the child nodes of E and H, which contains the values of "TXT1" and "TXT2" respectively.

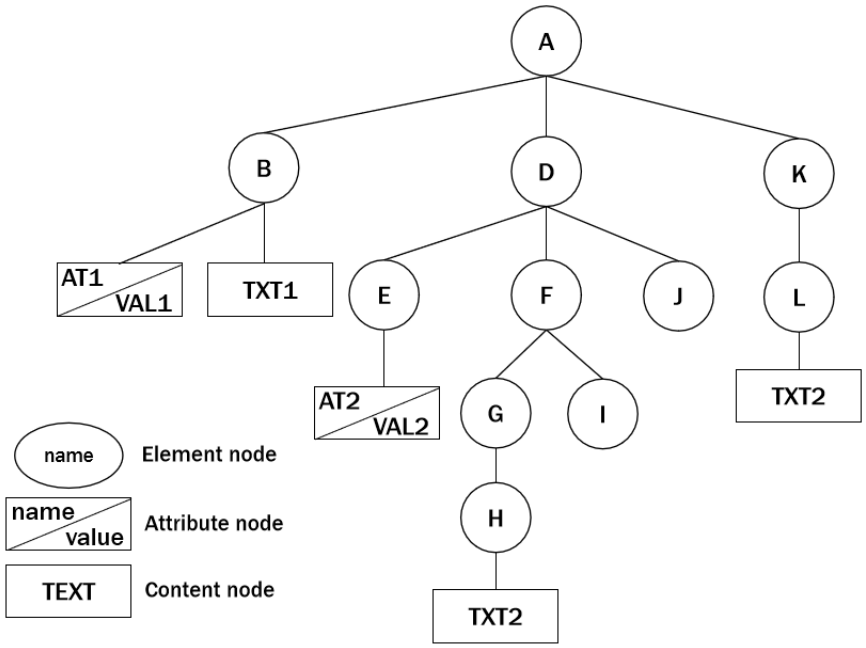


Figure 2.1: An example of XML tree with nodes of three node types.

2.2 XPath

2.2.1 Definition

XPath is an XML query language used for selecting parts of an XML document [95]. XPath queries are represented in path expressions. Each path expression contains one or more *location steps*.

An important aspect we need to consider when we process an XML document is the relationship between a pair of nodes, which is *axis* in XPath. There are 12 axes supported in this study, including `child`, `descendant`, `parent`, `ancestor`, `descendant-or-self`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, `preceding-sibling` and `attribute`. Note that `attribute` is different from the the other axes, because it relates to attribute nodes, while the other axes relate to element nodes. Content nodes can be selected by using function `text()`.

Let us take a look at an example shown in Figure 2.2. In the figure, we uses node F as the current node to demonstrate these axes. F has a parent D; E on the left side as a preceding-sibling and I on the right side as a following sibling, two children: G and

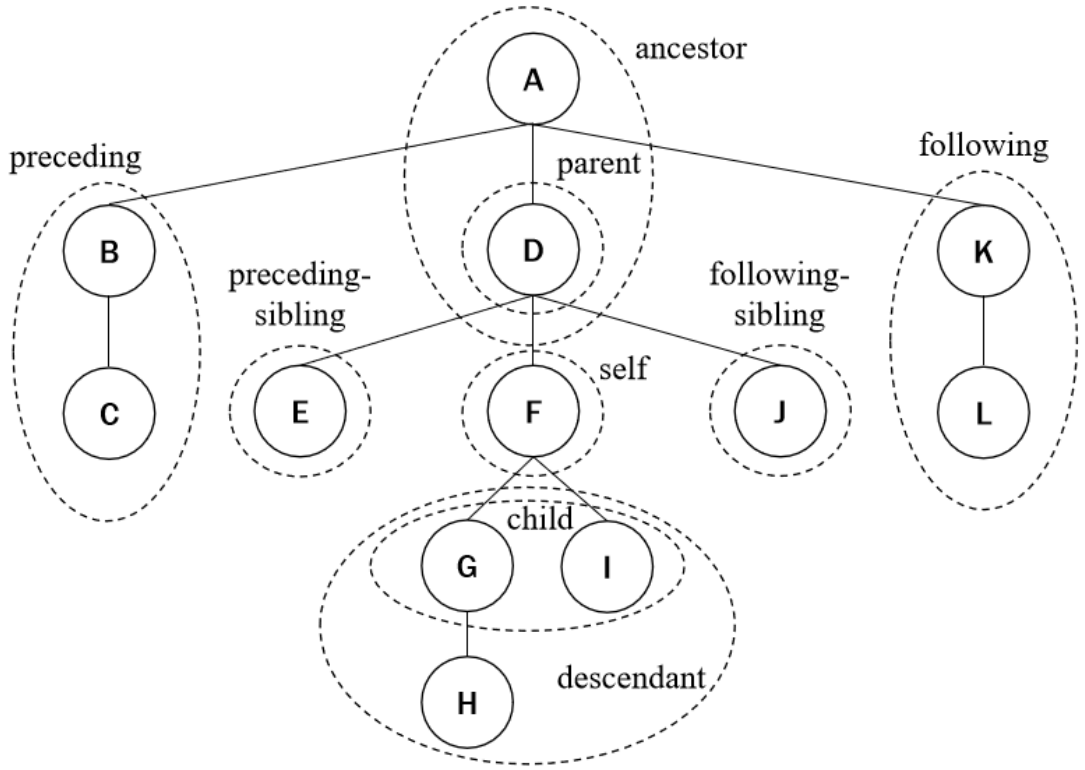


Figure 2.2: An example of node relationships.

H, one descendant I, one preceding B, and two followings: nodes K and L. Please be noted that a pair of nodes can have more than one relationships. For example, G and H are not only child nodes of F, but also descendant nodes of F.

A name test is used for selecting nodes. When a node match target axis, we apply nametest to filter nodes. If the name of a tag in an XML document is equal to the name test, the node is selected. XPath also defines a wilecard '*' that matches with any name.

A predicate written between "`[`" and "`]`" describes additional conditions to filter the matched nodes by using a path.

For example, given a query `/descendant::F[following-sibling::J]/child::H`, this XPath query has two steps where `descendant` and `child` are the axes, `F` and `H` are the name tests, and a predicate `child::H` is attached to the step `/descendant::F`.

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

Figure 2.3: Grammars of XPath queries used for partial tree

2.2.2 Evaluating XPath query

There different ways of evaluating an XPath queries. We give one way to demonstrate how an XPath is evaluated. In this way, we process an XPath query step by step so we can easily understand how the query works. We start from the first step and evaluate each location step in order. When meeting a predicate, we process it as a filter to rule out unmatched nodes. Let us continue to use the query "`/descentdant::F[following-sibling::J]/child::H`" for demonstration.

The evaluation of this query starts from the first step. By evaluating `/descendantt::F`, we retrieve all the descendant nodes with name `F` from the root of the XML tree. Since this step has a predicate, we will check whether the selected `F`s have a following-sibling node `J`. If so, the node is selected. For the first step, node `F` satisfies the step, thus being selected. Next, we select children of `F`, whose name is `H`. The result of the query is a set of nodes `H`, each of which has its parent `F` and at least one sibling `G` as its parent's following-sibling. The grammar of XPath used in this study for our novel tree structure partial tree (See Chapter 5) is listed in Figure 2.3.

2.3 XQuery

XQuery is also query language for processing XML documents compared with XPath. It supports variables, loops and more functions. It can integrate XPath lan-

guage into it. When an XPath query is integrated in an XPath expression, the XPath query still returns exactly the same result as in XPath language. Thus, XQuery is more expressive and powerful (also more complicated) than XPath.

Let us use consider the following expression as an example. Note that, we used XQuery 3.1 [9] in this study.

```
for $x in doc("books.xml")/bookstore/book
return $x/title
```

In this expression, we can define a variable `$x` to be used for a `for` loop. In this loop, an XML document, namely "books.xml", is searched fro the XPath query `/bookstore/book` and the result of the XPath query is stored in `$x`. Lastly, one more step `book` is evaluated on `$x` and the result of it is returned.

Chapter 3

Related Work

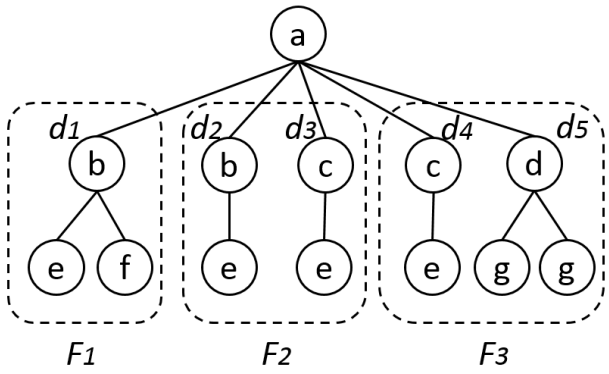
We discuss related work in three related fields, XML fragmentation, parallel XML processing and XML database Techniques.

3.1 XML Fragmentation

Fragmentation is a way of dividing an XML document into multiple smaller fragments denoting well-formed pieces of an XML document [89] by physical changes that split and separate the document so that these fragments can be allocated to multiple computational nodes [25]. It is worth noting that the term *fragment* refers to a piece of XML document containing well-formed strucuture. It is generally the premise of data-parallel computation algorithms. When process large XML documents, it is a natural way to reduce the size of XML documents processed at a time, so that we can process them with more computational nodes in parallel to boost the perform of parsing and querying. For these reasons, fragmentation has been intensively studied [14, 17, 19, 23**?**]. In this section, we discuss two fragmentations that closely relate to our study.

3.1.1 Horizontal and Vertical Fragmentation

There are two fragmentations defined by Kling et al. [59], who modeled fragmentation as horizontal and vertical in terms of XML schema [8].



(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 3.1: An example of horizontal fragmentation and the schema.

Horizontal Fragmentation

Horizontal fragmentation divides a document into multiple fragments and each fragment follows the same schema as that of the original XML document. When fragments follow the same schema, they usually have similar structure. Let us take the tree in Figure 3.1 as an example. We divide the tree into five fragments, i.e. d_1, d_2,..., d_5 in the dotted rectangles. According to the schema, a node 'b' has a single 'e' followed by one or zeor 'f'; a node 'c' has exact one 'e'; a node 'd' can have zero or multiple node 'g's. Thus, all the subtrees follow the schema in Fig. 3.1(b). Note that all the root nodes of fragments are at the same level with the same parent and ancestors. It is important feature that makes it possible to parallellize the evaluation of the same query on the fragments.

We then consider the whole document as a simple collection of fragments. Since the fragments follow the same schema, the queries can be evaluated on them by considering the schema. Horizontal fragmentation is rather straightforward and thus widely used in parallel XML processing [11, 21, 72]. In our study, we exploit and extend the horizontal fragmentation for a size-balanced division and a proposal distributed-memory environments.

Vertical Fragmentation

Vertical fragmentation, on the other hand, is a fragmentation but at different levels, where these fragments does not need to follow the same schema, such that the fragmetation can focus more on the clarification of the data store in the XML document rather than the structure of the data. Thus, this fragmentation is specially useful and works well for parallel XML processing in case XML data are integrated from different sites or organizations [36, 58].

3.1.2 Ad-hoc Fragmentation

In comparison with horizontal and vertical fragmentation, ad-hoc fragmentation does not consider the schema of XML documents. The partial tree (see Chapter 5) in our study is a fragmentation that does not need the schema of XML documents, thus we consider this study can be categorized as ad-hoc fragmentation.

A common ad-hoc fragmentation is called *hole-filler* model, where holes are portals used for connection of fragments and fillers are fragments that can connect to holes according to the structure of the original whole tree. Bose et al. proposed a fragmentation model for steam data [24] and a system called Xfrag [23]. In their studies, an XML document is divided into multiple sub documents, each of which has one or more holes to connect other sub document by the fillers. Another related work [61] proposed a hole-filler model over streamed XML fragments, which is particularly designed for good memory-efficiency. Nomura et al. [79] and Cong et al. [36] adopted a fragment that contains original nodes and hole nodes, where a hole node represents a link to a missing subtree, and represented the whole document as a tree of fragments. Their approaches decouple dependencies between evaluations on fragments so as to perform queries on them in parallel. Kawamura et al [56] proposed a fragmentation algorithm by using a sized-balanced algorithm to binary trees with an implementation on SAX. In our study, when we divide an XML document into fragments, a node may be divided into many nodes and is distributed on multiple fragments. We use links to represent the connections among these nodes of the same one. Although we did not use holes

and fillers, this way can be considered as ad-hoc fragmentation, since the links work exactly for the same purpose.

3.2 Parallel XML Processing

Many existing studies address the topic of parallel XML processing [21, 28, 70, 74, 84, 87]. We discuss parallel XML processing in this section.

3.2.1 Tree Reduction and Accumulation

There are some existing ideas about tree reduction, which can accelerate processing on XML trees by an efficient form or representation of trees. Kakehi et al. [55] showed a parallel tree reduction algorithm to with its start point to use serialized forms of trees. Based on the same idea, an improved algorithm was developed by Emoto and Imachi [41]. They restructured the algorithm for tree reduction computations, which can be used to implement various tree computations such as XPath queries. Tree accumulation is a common process that accumulates data placed in tree nodes according to their tree structure. Sevilgen et al. [87] who developed a simpler version of tree accumulations over the serialized representation of XML trees. Matsuzaki and Miyazaki [75] developed a parallel tree accumulation algorithm for binary trees by exploiting a tree division technique.

3.2.2 XML Streaming

Stream processing is a possible approach for (parallel) online data analysis. Parallel algorithms have been studied to accelerate stream processing of large XML data. For example, XMLTK [15] is an XML stream processing tool designed for scalable XML querying. Y-Filter [98] applies multiple queries in parallel to a stream of XML data. Among these studies, Ogden et al. [80] achieved the highest throughput, 2.5 GB/s, based on the parallel pushdown transducer. Although it is the fastest and thus is faster than our implementation of partial tree, which was 1 GB/s, the class of queries we support is still more expressive than that of parallel pushdown transducer, which does

not support order-aware queries.

3.2.3 XML Parsing

XML Parsing is a process of creating an XML tree from reading an XML document. [83, 93] focused on XML parsing, which is related to our parsing algorithm. The parallel XML parsing is based on serialized text. The plain text of an XML document is divided into multiple chunks to be parsed in parallel. Yinfei et al. [84] developed an algorithm for parsing the XML data in parallel without any sequential preparing phase. Based on parallel XML parsing, we propose partial tree that can achieve load-balance by dividing an XML document into many smaller fragments in similar sizes. A similar idea was introduced by Choi et al. [33] in which they added labels to construct a well-formed tree from a chunk in a preparing phase. The main advantage of our text-based fragmentation is that we can easily achieve load-balance by dividing an XML document into equal-sized chunks and then process them over distributed file systems, such as Hadoop Distributed File System [22].

3.2.4 Parallel Processing of queries

Parallel XML processing has been actively studied especially after a well-known paper had been presented by Bordawekar et al. [21], which is important to our study. The paper proposes three strategies for XPath queries in parallel: data partition strategy, query partition strategy, and hybrid partition strategy. Before this paer, there were some studies in the parallel programming community from 1990's. Skillicorn developed a set of parallel computational patterns for trees called tree skeletons, and showed they can be used for processing structured documents [88]. The main idea in parallelizing XPath queries was to convert XPath queries into (tree) automata [35], and then compute automata in parallel with tree skeletons. This idea was extended to support a larger class of XPath including `following-sibling` by Nomura et al. [79]. Some studies[60, 83, 98] focus on XPath queries implemented in a shared-memory environment. In [40], an XPath query was processed in a forward and downward manner. In contrast, our research can support backward and upward queries as well. Liu et

al. [67] developed a parallel version of structural join algorithm. The study [96] focuses processing a locality-aware partitioning in parallel database systems. Cong et al. [36] formalized parallel processing of XPath queries using the partial evaluation technique: the idea existing behind their partial evaluation is similar to automata. Compared with the above studies, the most important feature of our study is that we can support all axes of XPath queries in a distributed-memory environment.

MapReduce [39] is a promising parallel programming model for large-scale XML processing, running on clusters of commodity computers. It is suitable for persuing good scalability as the size of XML data increases very rapidly. Hadoop [92], a porpular implementation of MapReduce, is a common infrastructure for large-scale data processing, and to parallel streaming [54, 80]. There have been several studies in this direction [16, 36, 38, 41, 75] for parallel XML processing. One early work proposed by Choi et al. [32] called HadoopXML, processes XML data in parallel by applying SAX [3] for each XML chunk. Apart from this work, most of the existing MapReduce-based frameworks support a small subset of XPath, such as `child` and `descendant` axes with predicates [11, 27, 37, 38]. Instead, they extend the expressiveness by the support for some query functionalities (subsets of XQuery). To cope with the problem of absolute performance of MapReduce, there is a few work to use similar but more efficient frameworks, such as Apache Flink [27], Spark [1] etc.

3.3 XML Database Techniques

3.3.1 Indexing and Labeling Schemes

Indexing is a common database technique to accelerate the access of specific data by using some characteristic information of data so that the access to the data can be done immediately. Although XML databases are different from relational databases, it is still available to exploit indexing in XML databases. However, due to the tree structure, it is a challenge to create efficient indexing scheme for XML documents. In 2004, O'Neil et al. [81] proposed an indexing scheme called ORDPATH that can natively supports XML data type in a relational database. This index makes it possible to process XML

queries inside the database with downward XPath queries and allows update operations. Since this length of this index increases with respect to the size of XML documents, the length will be greatly increased in case the XML documents will be very large. Pal et al. [82] studied how to improve the query performance by introducing two indexes to nodes and values in SQL Server. Li et al [63] improved ORDPATH by reducing the length of ORDPATH index when inserting. Min et al. [77] proposed an efficient labeling scheme, called EXEL, which incurs no re-labeling of nodes when inserting nodes. Finis et al [42] Proposed an idea mainly on how to maintain and query hierarchical data at a high rate of complex, possibly skewed structural updates. Besides these index schemes, there are also some studies concerning specific types of trees, such as [31, 46, 51], which examined the differences in indexing trees, including B+-tree, R-tree, and XR-tree. BaseX[10] is a state-of-the-art XML database engine. It creates many different types of indexes for an XML database. For example, it uses Structural Indexes for structural information, Value Indexes for content, attribute values etc. It also allows user to define their own indexes for special purpose. In this thesis, by considering the above studies, a new index scheme is designed for representing partial tree particularly for processing large XML data in distributed-memory environments. For example, we use the start position and end position of an elementat in the original XML document as index to determine the document order.

3.3.2 Joins Algorithms

Join processing is central to database implementation [45].

Structural join [12] is mostly based on numbering indexing[13], which numbers a nested intervals on nodes and is commonly used in XML and other database applications [48, 97]. By using the information of start position, end position and level of each node, the parent-child and ancestor-descendant relationships of nodes can be determined by a merge join on two lists of nodes. In 2001, a study [64] proposed three join algorithms for processing XML queries, which were similar to structural join. In 2002, Quanzhong Li et al. first proposed structural join in [12]. Jiang et al. [51] improved the structural join with a novel tree structure, called XR-tree, which is suitable for

identifying the descendants of a given node with optimized worst case I/O cost. Liu et al. [67] first applied structural join in parallel over shared-memory environments.

Twig join is a subset of XPath language, which support child/descendant axes with predicates. It is also commonly used for matching a part of an XML documents [44, 53, 68, 69]. In twig join, a query is represented as a twig pattern, and then is searched on the target XML document. One of the early twig joins was [26]. In the paper, a holistic twig join algorithm, called TwigStack was proposed for matching an XML query. There are also variants of twig joins then developed [30, 86]. In 2009, Machdi et al. [73] implemented the idea in parallel on multiple cores and in 2012 Choi et al. [32] studied the twig joins on Hadoop in a parallel manner.

Chapter 4

Parallelization of XPath Queries on Top of BaseX

There was a practical and promising study by Bordawekar et al. [20] for the parallelization of XPath queries in 2009. In this study, three partitioning strategies were presented: data partitioning, query partitioning and hybrid partitioing strategies, which allows us to parallelize XPath queries by partitioning an XPath query into sub-queries and evaluating them in parallel on XML trees.

However, since this study was based on an XSLT processing and the hardware also changed a lot in term of number of cores, it is thus not clear to the following questions: *(1) Whether and how can we apply their partitioning strategies to XML database engines? (2) How much speedup can we achieve on large XML documents?.*

To answer the above two questions, we introduce our implementations on top of a state-of-the-art XML database engine BaseX by reviving and extending Bordawekar et al's study. We propose implementations on the base of the original idea from the paper by exploiting its optimizations. We experimentally demonstrate the performance on top of BaseX with the two large XML documents. The experiment results showed that it is possible to obtain significant speedups by simply rewriting queries into sub-queries and parallelizing the evaluation of them on top of an XML database engine over gigabytes of XML documents, without needs to modify source code of the engine.

4.1 BaseX: An XML Database Engine

To begin with, we give a brief introduction to BaseX, which is both a state-of-the-art XML database engine and an XQuery/XPath 3.1 processor (refer to the official documentation [10] for more details). BaesX provides many features in processing XML data sets. The following are BaseX's features particularly important for our study:

- Full support of XQuery 3.1, especially arrays and XQuery Update Facility;
- XQuery extension for database operations, especially index-based random access;
- XQuery extension for full-text operations, especially `ft:tokenize`;
- Support of in-memory XML databases;
- Query optimization based on various indices;
- Support of concurrent transactions in the server mode.

The first three items are concerned with the expressiveness of queries and the rests are concerned with performance.

The most important (practically essential) feature to our implementations is index-based random access to nodes of an XML tree stored as a database in BaseX. BaseX offers indices that enable us to access any node in constant time. The *PRE* index, which denotes the position of a node in document order, brings the fastest constant-time access in BaseX. Function `db:node-pre` returns the PRE value of a given node and function `db:open-pre` returns the node of a given PRE value. For example, given a BaseX database "exampledb", as shown below.

```
1<books>
  2<book>
    3<name>4XML</name>
    5<author>6Jack</author>
  </book>
</books> ,
```

Note that a left superscript denotes a PRE value in the code. Now, consider the following query.

```
for $node in db:open(‘‘exampledb’’)/books/book
return db:node-pre($node)
```

The query selects `book` in "exampledb" and the final value of `$node` is `<book>... </book>`. Then, after applying the `db:node-pre` function the final result is 2. PRE values are well-suited for representing intermediate results of XPath queries because a PRE value is merely an integer that makes it efficient to restart a tree traversal with PRE values. Letting a PRE value be 2 on the 'exampledb', we use the following query as an example.

```
for $pre in db:open-pre(‘‘exampledb’’, 2)/author
return $pre
```

The `db:open-pre` takes the database and a PRE value as arguments to locate the `book` node. Then it selects the `author` of `book`. The final result is `<author>Jack</author>`.

Arrays are useful for efficiently implementing block partitioning. On BaseX, the length of an array is returned in constant time and the sub-array of a specified range is extracted in logarithmic time[1]. XQuery Update Facility over in-memory databases strongly supports efficient use of temporary databases for holding results of queries. Function `ft:tokenize`, which tokenizes a given string to a sequence of token strings, can implement deserialization of sequences/arrays efficiently.

Lastly, BaseX can work in a client-server mode. A BaseX server can handle concurrent transactions requested from multiple BaseX clients with multiple threads depending on the server's configurations. Read transactions that do not modify data, such as XPath queries, are executed in parallel without waiting or using locks in BaseX.

4.2 Implementing Data Partitioning with BaseX

In this section, we describe our two implementations for data partitioning strategy on top of BaseX, namely the client-side implementation and the server-side implemen-

[1]In fact, both sequences and arrays on BaseX are implemented with finger trees and therefore the corresponding operations on sequences have the same cost.

tation.

Data partitioning is to split a given query into a prefix query and a suffix query, e.g., from q_1/q_2 to prefix q_1 and suffix q_2, and to run the suffix query in parallel on each node of the result of the prefix query. The results of suffix queries are concatenated in document order to form the final result.

Our implementations are Java programs that involve a BaseX client. They spawn P threads (usually P is no more than the number of physical cores) and create a connection to a BaseX server for each thread so as to run multiple queries in parallel after a prefix query. Merging P partial results in the form of string is sequentially implemented. The main difference between the client-side and the server-side implementations is the way how the results of prefix queries are handled. In the rest of this section, we describe them by using XM3(a) shown in Table 4.1 as a running example, assuming input database to be named "xmark.xml".

4.2.1 Client-side Implementation

The client-side implementation is a simple implementation of data partitioning strategy with database operations on BaseX. It sends the server a prefix query to be executed and the PRE values of matched nodes are returned. The following XQuery expression is used for the prefix query of XM3(a).

```
for $x in db:open(‘‘xmark’’)/site//open_auction
return db:node-pre($x)
```

Let this prefix query return sequence `(2, 5, 42, 81, 109, 203)`. Letting $P = 3$, it is block-partitioned to `(2, 5)`, `(42, 81)`, and `(109, 203)`, each of which is assigned to a thread. To avoid repetitive ping-pong between a client and the server, we use the following suffix query template:

```
for $x in ⟪sequence of PRE⟫
return db:open-pre(‘‘xmark.xml’’, $x)/bidder[last()] ,
```

Figure 4.1: List of XPath queries and their partitioning, where pre and suf mean prefix query and suffix query respectively

Key	Query
XM1	`/site//*[name(.)="emailaddress" or name(.)="annotation" or name(.)="description"]`
XM1(a)	pre = `/site/*`, suf = `descendant-or-self::*[name(.)="emailaddress" or name(.)="annotation" or name(.)="description"]`
XM2	`/site//incategory[./@category="category52"]/parent::item/@id`
XM2(a)	pre = `/site//incategory`, suf = `self::*[./@category="category52"]/parent::item/@id`
XM2(b)	pre = `/site/*`, suf = `descendant-or-self::incategory[./@category="category52"]/parent::item/@id`
XM2(c)	pre = `db:attribute("xmark10", "category52")`, suf = `parent::incategory[ancestor::site/parent::document-node()]/parent::item/@id`
XM3	`/site//open_auction/bidder[last()]`
XM3(a)	pre = `/site//open_auction`, suf = `bidder[last()]`
XM3(b)	pre = `/site/*`, suf = `descendant-or-self::open_auction/bidder[last()]`
XM3(c)	pre = `/site/open_auctions/open_auction`, suf = `bidder[last()]`
XM4	`/site/regions/*/item[./location="United States" and ./quantity > 0 and ./payment="Creditcard" and ./description and ./name]`
XM4(a)	pre = `/site/regions/*`, suf = `item[./location="United States" and ./quantity > 0 and ./payment="Creditcard" and ./description and ./name]`
XM4(b)	pre = `/site/regions/*/item`, suf = `self::*[./location="United States" and ./quantity > 0 and ./payment="Creditcard" and ./description and ./name]`
XM4(c)	pre = `db:text("xmark10", "Creditcard")/parent::payment`, suf = `parent::item[parent::*/parent::regions/parent::site/parent::document-node()][location = "United States"][0.0 < quantity][description][name]`
XM5	`/site/open_auctions/open_auction/bidder/increase`
XM5(a)	pre = `/site/open_auctions/open_auction/bidder`, suf = `increase`
XM5(b)	pre = `/site/open_auctions/open_auction`, suf = `bidder/increase`
XM6	`/site/regions/*[name(.)="africa" or name(.)="asia"]/item/description/parlist/listitem`
XM6(a)	pre = `/site/regions/*`, suf = `self::*[name(.)="africa" or name(.)="asia"]/item/description/parlist/listitem`
XM6(b)	pre = `/site/regions/*[name(.)="africa" or name(.)="asia"]/item`, suf = `description/parlist/listitem`
DBLP1	`/dblp/article/author`
DBLP1(a)	pre = `/dblp/article`, suf = `author`
DBLP2	`/dblp//title`
DBLP2(a)	pre = `/dblp/*`, suf = `self::*/sef-or-descendant::title`

where ⟪sequence of PRE⟫ is a placeholder to be replaced with a concrete partition, e.g., `(42, 81)`. Each thread instantiates this template with its own partition and sends the server the instantiated query.

4.2.2 Server-side Implementation

A necessary task on processing the results of a prefix query is to block-partition them. The client-side implementation simply processes it on the client side. In fact, we can also implement it efficiently on the server side by utilizing BaseX's features.

Firstly, we prepare an in-memory database named ``tmp'' and initialize it with `<root> </root>`, which is a temporary database for storing the results of a prefix query. The prefix query is `/site//open_auction`, which selects all `open_auction` and returns the PRE values of matched nodes. The results of the prefix query are stored on the server side. It is implemented as follows:

```
let $P := ⟪number of partitions⟫
let $arr := array { for $x in db:open(``xmark'')/site//open_auction
                    db:node-pre($x) }
for $i in 1 to $P
return insert node element part { $block_part($i, $P, $arr) }
        as last into db:open('tmp')/root
```

In the code, ⟪number of partitions⟫ denotes a placeholder to be replaced with a concrete value of P and `$block_part($i, $P, $arr)` denotes the `$i`-th sub-array of `$P`-block partitioned `$arr`. With the array operations of extracting length and sub-array, `$block_part` is implemented in logarithmic time.

In the example case used earlier, ``tmp'' database results in the following:

[1]<root>
  [2]<part>[3]2 5</part>[4]<part>[5]42 81</part>[6]<part>[7]109 203</part>
 </root> ,

where a left superscript denotes a PRE value. Note that its document structure determines the PRE value of ith partition to be $2i + 1$.

A suffix query is implemented with deserialization of a partition as follows:

```
for $x in ft:tokenize(db:open-pre(''tmp'', ⟪PRE of partition⟫))
return db:open-pre('xmark', xs:integer($x))/bidder[last()]) ,
```

where ⟪PRE of partition⟫ denotes a placeholder to be replaced with the PRE value of a target partition.

In most case, the server-side implementation is more efficient because communication data between clients and a server except for output is reduced to a constant size.

4.3 Implementing Query Partitioning with BaseX

In this section, we describe our implementation of query partitioning strategy on top of BaseX. The implementation of query partitioning strategy is also a Java program that involves a BaseX client. This implementation is relatively simpler than that of data partitioning. It has two phases, a parallel evaluating phase and a merging phase. In the first phase, a query is divided into multiple sub-queries, which are then executed by a BaseX server in parallel. In the second phase, the results of all sub-queries are merged together into a whole as the final result. In the following paragraphs, we focus the parallel evaluating phase of query partitioning in detail.

There are two ways of partitioning an XPath query in the parallel evaluating phase: position-based partitioning and predicate-based partitioning. Position-based partitioning is to divide the query by the `position` function on a specific location step. It is based on the partition of children of a node in particular positions such that these children are divided into multiple groups in order. Then, we evaluate sub-queries on these groups of nodes in parallel. Let us take XM5 as example. The number of chil-

dren `/site/open_auctions` on 'xmark10.xml' is 120000, which can be obtained from statistics of the database by using *count* function in BaseX. Letting P = 3, then we can use the `position` function to divide the query into three sub-queries as follows:

```
/site/open_auctions/open_auction[position()=1 to 40000]/bidder/increase
/site/open_auctions/open_auction[position()=40001 to 80000]/bidder/increase
/site/open_auctions/open_auction[position()=80001 to 120000]/bidder/increase
```

These three sub-queries cover exactly the same amount of nodes on the target XML document as that of the original query and returns in turn the same results. Note that since the query is divided by the `position` function in document order, the results are also in the document order. Thus, we can simply merge the results to form the final result.

As for the predicate-based partitioning, it is not so promising. This is because it often takes a lot extra time to merge the results of sub-queries. For example, given a query `/q1[q2 or q3]` that is divided into `q1[q2]` and `q1[q3]`, we first retrieve results from both queries. Since the operation in the predicate is an '`or`', we need to perform an ordered union to the results of the two sub-queries. Since the nodes in the results should be in the document order, we have to sort the merged nodes after merging. However, there are two difficulties to merge them. First, we need to find a way to determine the order of the merged nodes. Second, the merging process itself is time-consuming. Therefore, we do not focus on the partitioning in our study.

We also extend the query partitioning strategy by partitioning the children with different names, i.e. the partitioning is based on branches. Since it is often to have different node names at the first level, it is worth studying the way of partitioning. We exploit the structure of the input XML document to evaluate sub-queries on the branches of a node by walking through its children in parallel. Let us take XM1(d) on '`xmark10.xml`' as an example. Since the root of the document has only six nodes: `regions`, `categories`, `catgraph`, `people`, `open_auctions`, `closed_auctions`, we can make the input query into six corresponding sub queries. Given the first child `regions` of the root, we make the corresponding subquery as

`/site/regions/descendant-or-self::incategory.../@id`,

Figure 4.2: List of XPath queries used for query partitioning, where [pos] denotes position-based partitioning and {name} denotes branch-based partitioning.

XM1	`/site//*[name(.)="emailaddress" or name(.)="annotation" or name(.)="description"]`
XM1(d)	`/site/*[pos]/descendant-or-self::*[name(.)="emailaddress"` or name(.)="annotation" or name(.)="description"]—
XM1(e)	`/site/{name}/descendant-or-self::*[name(.)="emailaddress" or name(.)="annotation" or name(.)="description"]`
XM2	`/site//incategory[./@category="category52"]/parent::item/@id`
XM2(d)	`/site/regions/*[pos]/item/incategory[./@category="category52"] /parent`
XM2(e)	`/site/regions/{name}/item/incategory[./@category="category52"] /parent`
XM3	`/site//open_auction/bidder[last()]`
XM3(d)	`/site/open_auctions/open_auction[pos]/bidder[last()]`
XM4	`/site/regions/*/item[./location="United States" and ./quantity > 0 and ./payment="Creditcard"and ./description and ./name]`
XM4(d)	`/site/regions/*[pos]/item[./location="United States" and ./quantity > 0 and ./payment="Creditcard and ./description and ./name]`
XM4(e)	`/site/regions/{name}/item[./location="United States" and ./quantity > 0 and ./payment="Creditcard" and ] ./description and ./name`
XM5	`/site/open_auctions/open_auction/bidder/increase`
XM5(d)	`/site/open_auctions/open_auction[pos]/bidder/increase`
XM5(e)	`/site/open_auctions/open_auction/bidder[pos]/increase`
XM6	`/site/regions/*[name(.)="africa" or name(.)="asia"] /item/description/parlist/listitem`
XM6(d)	`/site/regions/*[pos][name(.)="africa" or name(.)="asia"] /item/description/parlist/listitem`
XM6(e)	`/site/regions/\{name\}[name(.)="africa" or name(.)="asia"] /item/description/parlist/listitem`
DBLP1	`/dblp/article/author`
DBLP1(d)	`/site/regions/*[pos][name(.)="africa" or name(.)="asia"] /item/description/parlist/listitem`
DBLP2	`/dblp//title`
DBLP2(d)	`/dblp/{name}/titlem`

which is to be evaluated through only the first branch of the root and selects `@id`s that matches the query in that branch. The six corresponding sub-queries cover exactly the same part of the original query and return the same results. Because the children of the root are in document order, the results are also ordered as long as the results are merged in the same order.

4.4 Integration with Query Optimization

As mentioned in Section 4.1, BaseX is equipped with a powerful query optimizer. Some queries can be optimized to reduce execution time. For example, XM3 will be optimized as follows.

```
/site/open_auctions/open_auction/bidder[last()]
```

This optimization is on the basis of the path index, which brings knowledge that `open_auction` exists only immediately below `open_auctions` and `open_auctions` exists only immediately below `site`. Because the step of descendant-or-self axis, `//open_auction`, is replaced with two child steps (`/open_auctions/open_auction`), the search space of this query has been significantly reduced. Note that a more drastic result is observed in XM2, where the attribute index is exploited through function `db:attribute`.

When data partitioning strategy converts a given query to separate ones, it may affect the capability of BaseX in query optimization. For example, the suffix query of XM3(b) is not optimized to the corresponding part of optimized XM3 because BaseX does not utilize indices for optimizing queries starting from nodes specified with PRE values even if possible in principle. Most index-based optimizations are limited to queries starting from the document root. This is a reasonable design choice in query optimization because it is expensive to check all PRE values obtained from the evaluation of a prefix query. However, we do not have to check all PRE values that specify the starting nodes of the suffix query because of the nature of data partitioning, of which BaseX is unaware. This discord between BaseX's query optimization and data partitioning may incur serious performance degradation. However, such discord does not occur in query partitioning strategy, because the queries are always applied to the root of the whole tree no matter how the queriesis partitioned.

A simple way of resolving this discord is to apply partitioning strategies to the BaseX-optimized query. Partitioning strategies are applicable to any multi-step XPath query in principle. Even if an optimized query is thoroughly different from its original query as in XM2, it is entirely adequate to apply data/query partitioning strategies to

the optimized query, forgetting the original. In fact, XM2(c)–XM4(c) are instances of such data partitioning after optimization.

The simplicity of this coordination brings two big benefits. One is that we are still able to implement partitioning strategies only by using BaseX's dumps of optimized queries without any modification on BaseX. The other is that it is very easy to implement partitioning strategies into compilation in BaseX; we can just add a data/query-partitioning pass after all existing query optimization passes without any interference.

We must also note that although BaseX's query optimization is very powerful, it is not unusual to improve time complexity. For example, path index enables pruning traversal of descendants and attribute index enables instant access to the nodes that have a specific attribute name or value. With aggressive constant propagation, BaseX exploits most constants including database metadata and PRE values found in a given query for query optimization. Prevention of spoiling it is therefore of crucial importance for performance.

4.5 Experiments and Evaluations

In this section, we introduce the experiments conducted on two datasets for evaluating the performance of our implementations and report the experiment results.

4.5.1 Experiment Settings

We have conducted several experiments to evaluate the performance of our implementations of parallel XPath queries. All the experiments were conducted on a computer that equipped with two Intel Xeon E5-2620 v3 CPUs (6 cores, 2.4GHz, Hyper-Threading off) and 32-GB memory (DDR4-1866). The software we used ware Java OpenJDK ver. 9-internal (64-Bit Server VM) and BaseX ver. 8.6.4 with minor modifications to enable TCP_NODELAY, which is used to improve TCP/IP networks.

We used two larege XML documents for the experiments: `xmark10.xml` and `dblp.xml`. The dataset `xmark10.xml` is an XMark dataset [94] generated with the pa-

rameter 10, which was of 1.1 GB and had 16 million nodes. The root of the XMark tree has six children `regions`, `people`, `open_auctions`, `closed_auctions`, `catgraph`, and `categories`, which have 6, 255000, 120000, 97500, 10000, and 10000 children, respectively. The dataset `dblp.xml` was downloaded on Febrary 13, 2017 from [62], which is an open bibliographic information database on major computer science journals and proceedings. It sizes 1.85 GB and has 46 million nodes. The root has 5.5 million nodes of eight different names, including `article`, `book`, `incollections`, `inproceedings`, `masterthesis`, `phdthesis`, `proceedings`, `www`. We used the XPath queries shown in Table 4.1 for data partitioning and Table 4.2 for query partitioning. Those queries are originally from [21] but modified or optimized for BaseX's engine. We measured execution times from sending queries to BaseX until we obtained the whole serialized string as the result. The execution time does not include the loading time, that is, the input XML tree was loaded into memory before the execution of queries. To reduce the effect of fluctuations, we measured execution time for 51 times, and calculated their average after removing top-10 and bottom-10 results.

4.5.2 Evaluation on Implementations of Data Partitioning Strategy

In this section, we evaluate two of our implementations of data partitioning strategy. We also analysis the speedup and scalability of them.

Total Execution Time

Table 4.1 summarizes the execution times of the queries. The "orig t_o"column shows the time for executing original queries XM1–XM6 and DBLP1–DBLP2 with BaseX's `xquery` command. The "seq t_s" columns show the time for executing the prefix query and the suffix query with a single thread. The "par t_p" columns show the time for executing the prefix query with one thread and the suffix query with 12 threads (6 threads for XM1(a) and 8 threads for DBLP2(a)). The table also includes reference for the speedup of parallel queries with respect to original queries and the size of results of the prefix queries and the whole queries.

By using the pair of the prefix and suffix queries split at an appropriate step, we

Table 4.1: Summary of execution time of data partitioning

Key	orig t_o	client-side			server-side			Result size	
		seq t_s	par t_p	(t_o/t_p)	seq t_s	par t_p	(t_o/t_p)	prefix	final
XM1(a)	25796.64	27916.83	12392.80	(2.08)	28161.89	11484.99	(2.25)	54	994 M
XM2(a)	1.33	2996.85	959.18	(0.00)	1159.78	760.62	(0.00)	6.62 M	1.55 K
XM2(b)		1018.59	707.38	(0.00)	894.94	529.75	(0.00)	54	
XM2(c)		3.43	4.56	(0.29)	5.29	6.26	(0.21)	671	
XM3(a)	595.75	900.69	297.88	(2.00)	706.95	226.54	(2.63)	1.08 M	14.5 M
XM3(b)		1519.92	1148.42	(0.52)	1472.53	987.48	(0.60)	54	
XM3(c)		1029.85	308.67	(1.93)	723.44	297.31	(2.00)	1.08 M	
XM4(a)	798.16	1290.36	699.99	(1.14)	1241.34	559.32	(1.43)	49	26.4 M
XM4(b)		1786.89	564.82	(1.41)	1216.57	406.93	(1.96)	1.75 M	
XM4(c)		929.37	204.69	(3.90)	872.97	209.72	(3.81)	106 K	
XM5(a)	659.76	2311.56	751.65	(0.88)	1212.33	564.28	(1.17)	5.38 M	15.9 M
XM5(b)		1018.26	500.98	(1.32)	832.28	501.04	(1.32)	1.08 M	
XM6(a)	790.99	811.57	639.59	(1.24)	825.20	661.54	(1.20)	49	22.2 M
XM6(b)		875.33	189.20	(4.18)	810.15	190.67	(4.15)	183 K	
DBLP1(a)	3797.36	9138.82	2219.10	(1.71)	6152.72	1895.17	(2.00)	13.2 MB	133 MB
DBLP2(a)	9684.71	29389.93	9473.94	(1.02)	12789.19	5718.26	(1.69)	47.0 MB	356 MB

Table 4.2: Breakdown of execution time for client-side implementation

Key	prefix	suffix t^P						merge
		P=1	P=2	P=3	P=6	P=12	(t^1/t^{12})	
XM1(a)	4.63	27569.07	20756.44	20270.44	11758.00		(2.34)	287.08
XM3(c)	66.57	938.62	505.57	376.84	259.64	229.03	(4.10)	6.34
XM4(c)	14.64	895.02	550.90	399.07	215.49	172.66	(5.18)	9.12
XM5(b)	68.24	927.22	668.38	533.90	452.73	424.29	(2.19)	3.70
XM6(b)	17.00	842.94	488.21	360.93	194.28	157.65	(5.35)	8.17
DBLP1(a)	772.518	8412.663	5358.734	3512.645	2017.838	1413.355	(5.95)	33.222
DBLP2(a)	2006.868	29261.43	17381.72	12747	7843.271	7194.168	(4.07)	272.906

obtained speedups of factor two for XM1 and XM3, and factor of more than 3.5 for XM4 and XM6. The original execution time of XM2 was very short since BaseX executed an optimized query that utilized an index over attributes. By designing the parallel query XM2(c) based on that optimized query, the execution time of parallel query was just longer than that of original query by 5 ms. For the DBLP1(a) and DBLP2(b), the speedups are 1.71 and 1.02 on the client-side, while 2.00 and 1.69 on the server-side. Comparing the client-side and server-side implementation, we observed that the server-side implementation ran faster for most queries and performance differences were merely within the fluctuations even for the exceptions.

Breakdown of Execution Time

To investigate the execution time in detail, we executed parallel queries XM1(a), XM3(c), XM4(c), XM5(b), XM6(b) DBLP(a) and DBLP(a) with $P = 1$, 2, 3, 6, and

Table 4.3: Breakdown of execution time for server-side implementation

Key	prefix	suffix t^P						merge
		P=1	P=2	P=3	P=6	P=12	(t^1/t^{12})	
XM1(a)	5.09	27798.44	21155.57	20121.32	11047.99		(2.52)	192.27
XM3(c)	72.34	631.65	380.00	284.87	202.25	210.98	(2.99)	3.43
XM4(c)	16.20	840.24	530.57	376.44	198.55	170.17	(4.94)	10.46
XM5(b)	65.27	744.58	526.82	435.24	407.45	423.56	(1.76)	4.99
XM6(b)	17.22	776.99	450.99	323.49	176.92	157.47	(4.93)	5.98
DBLP1(a)	814.872	5298.603	2954.788	2092.14	1245.178	1039.821	(5.10)	40.475
DBLP2(a)	1911.724	13437.98	9223.007	6787.181	3812.572	3459.338	(3.88)	347.2

12 threads. Tables 4.2 and 4.3 show the breakdown of the execution time divided into three phases: prefix query, suffix query and merge. In these tables, the speedup is calculated with respect to the execution time of suffix queries with one thread.

From Tables 4.2 and 4.3 we can find several interesting observations. First, the execution time of prefix queries was almost proportional to their result sizes and almost the same between the two implementations. Comparing the two implementations, we can observe that the server-side implementation outperformed the client-side implementation in all suffix queries, where differences in merge were by definition within the fluctuations. These results suffice for concluding that the server-side implementation is, as expected, more efficient.

Next, we analyze the dominant factor of the performance gaps between the client-side and the server-side implementations. Although the performance gaps of prefix queries should be mainly the difference between sending data to clients on localhost and storing data into memory, it was not significant. Communication cost, which is our expected advantage of the server-side implementation, therefore did not explain the dominant factor of total performance gaps.

By examining the logs of the BaseX server, we have found that the dominant factor was parsing of suffix queries. Since the client-side implementation sends a suffix query of length linearly proportional to the result size of a prefix query, it can be long. In fact, the suffix query of XM3(c) for the 1-thread case was 1.1 MB and BaseX took 141.82 ms for parsing the query string. Sending and receiving a long query would not cost so much because localhost communication and local memory access were not so different in performance. Parsing is, however, more than sequential memory read like

deserialization that the server-side does. Parsing is essentially not streamable. Before finishing parsing a query, BaseX cannot start to evaluate it, whereas the deserialization in the server-side is streamable. We conclude that this difference in streamability was the dominant factor of the performance gaps between the client-side and the server-side.

For the DBLP queries, apart from the results that follow the observations from queries for XMark, We also notice that BaseX did not apply optimization on the descendant axis in DBLP2, which is caused by the structure of `dblp` dataset, which has 5515443 child nodes of eight unique names: `article, book, incollections, inproceedings, masterthesis, phdthesis, proceedings, www`. Every child of the root `dblp` represents a publication that has the information such as `title`, `author`, `year` etc. Since the paths are different (from example, a `title` could on the path of `/dblp/article/title` or `/dblp/book/title`), BaseX cannot apply the optimization that replaces descendant axis with the same child axes to all the matching nodes of `title`. In this case, query partitioning is useful to improve the query performance. We discuss this in Section 4.5.3.

Scalability Analysis

When we analyze the speedup of parallel execution, the ratio of sequential execution part to the whole computation is important because it limits the possible speedup by Amdahl's law. In the two implementations, the sequential execution part consists of the prefix query and merge. The ratio of the sequential execution part was small in general: more specifically, the client-side implementation had smaller ratio (less than 7%) than the server-side implementation had (less than 10%). In our implementation, the suffix queries were executed independently in parallel through individual connections to the BaseX server. The speedups we observed for the suffix queries were, however, smaller than we had expected. We also noticed that in some cases the execution time was longer with 12 threads than with 6 threads (for example, XM5(b) with the server-side implementation).

To understand the reason why the speedups of the suffix queries were small, we made two more analyses. Figure 4.3 plots the degree of load balance of the suffix

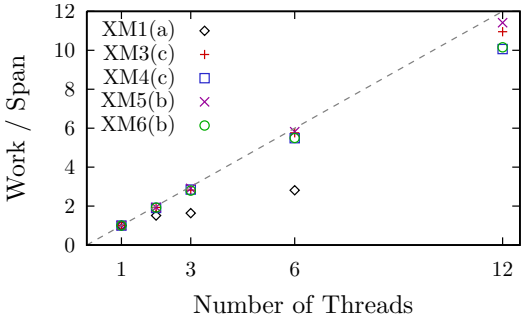


Figure 4.3: Load balance

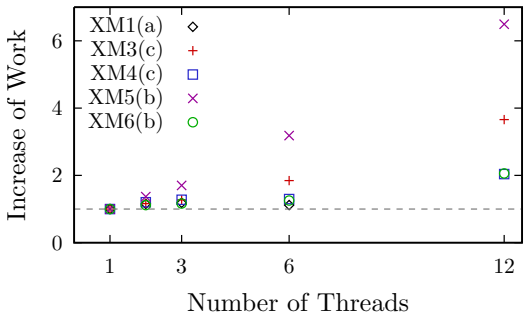


Figure 4.4: Increase of work

query, calculated as the sum of execution times divided the maximum of execution times. The degree of load balance is defined as $\sum t_i^p / \max t_i^p$, where t_i^p denotes the execution time of the ith suffix query in parallel with p threads. Figure 4.4 plots the increase of work of the suffix queries, calculated by the sum of execution times divided by that of one thread. The increase of work is defined as $\sum t_i^p / t_1^1$.

From Figure 4.3 and 4.4, we can observe the reasons of the small speedups in the suffix queries. First, when the prefix query returned a very small number of results (as for XM1(a)), the load of suffix queries was ill balanced. This was the main cause that the query XM1(a) had small speedups in the suffix queries. For the other cases, we achieved good load-balance until 6 threads, and the degrees of load-balance were more than 83% even with 12 threads, which means that load ill-balance was not the main cause of small speedups for those queries. Secondly, the increase of work was significant for XM5(b) and XM3(c), and it was the main cause that the queries XM5(b) and XM3(c) had small speedups. For the other queries, we observed almost no increase of work until 6 threads, but the work increased when 12 threads. Such an increase of work is often caused by contention to memory access, and it is inevitable in shared-memory multicore computers.

4.5.3 Evaluation on Implementation of Query Partitioning Strategy

Table 4.4 summarizes the total execution time of the queries. The "orig t_o" column shows the time for executing original queries XM1–XM6 and DBLP1–DBLP2 with BaseX's `xquery` command. The "seq t_s" columns show the time for executing all the

Table 4.4: Summary of total execution times(ms) of queries by query partitioning

Key	orig t_o	Implementation of Query Partitioning			Result size
		seq t_s	par t_p	t_o/t_p	Final
XM1(d)	25796.64	25326.472	11624.09	2.22	994M
XM1(e)		31561.45	11514.87	2.24	
XM2(d)	1.33	362.564	415.38	0.00	1.55 K
XM2(e)		4.05	1.26	1.06	
XM3(d)	595.75	754.439	146.64	4.06	14.5 M
XM4(d)	798.16	1098.432	516.98	1.54	26.4 M
XM4(e)		1163.919	523.62	1.52	
XM5(d)	659.76	869.089	288.21	2.29	15.9 M
XM5(e)		1536.161	1031.46	0.64	
XM6(d)	790.99	744.873	646.67	1.22	22.2 M
XM6(e)		739.449	640.57	1.23	
DBLP1	3797.36	6572.87	1618.51	2.35	133 M
DBLP2	9684.71	12524.70	6160.54	1.57	356 M

Table 4.5: Breakdown of execution time (ms)

Key	sub-queries						Merge
	P=1	P=2	P=3	P=6	P=12	t^1/t^{12}	
XM1(d)	25606.42	18583.66	18701.76	11420.36		2.24	203.73
XM2(d)	356.67	384.08	367.99	345.89	415.38	0.86	0.00
XM3(d)	540.14	327.03	241.33	159.63	141.82	3.81	4.82
XM4(d)	897.38	786.46	550.56	507.15		1.77	9.83
XM5(d)	670.15	415.35	314.01	284.40	282.14	2.38	6.07
XM5(e)	701.94	698.77	735.18	828.65	1025.09	0.68	6.37
XM6(d)	789.65	799.93	790.82	636.41		1.24	10.26
DBLP1(d)	4035.83	2609.61	1931.22	1567.33	1584.97	2.55	33.53

subquery one by one with a single thread. The "par t_p" columns show the time for executing a query with 1, 2, 3, 6, 12 threads depending on queries. The table also includes reference of the speedup of parallel queries with respect to original queries and the size of results of the original queries.

Total Execution Times and Speedups

From Table 4.4, we can see that for most of the queries we have obtained speedups and XM3(d) obtains the most speedup of a factor of 4.06. This means that we can accelerate the execution by query partitioning for these queries. We also notice that

there are two queries, on the contrary, which have been decelerated: XM2(d) and XM5(e), even with up to 12 threads.

Now, we explain the causes of the slowdown of the two queries.

For XM2(d), one obvious reason is that the original query takes too short time (only 1.33 ms), while the partitioned sub-queries take extra time for parallel execution and merge operation. However, there still a big gap between XM2(d) and XM2(e), i.e. XM2(e) takes quite less time than XM2(d) and is much close to the original query of XM2. The difference is caused by the sub-queries. For XM2(d), since XM2 is partitioned by the `position` function at the position right after `/site/regions/*`, it evaluates all the nodes that matches queries, thus taking over 500 ms to complete the query. While for XM2(e), it actually uses the attribute optimization so that the execution time has been greatly reduced. This is because the sub-queries of XM2(e) have complete paths. For example, one of its subquery is `/site/regions/africa/.../parent::item/@id`. Since the full path is contained in the subquires, BaseX can use the expression `db:attribute("xmark10", "category52")` to visit only the attribute nodes with the name "`category52`", avoiding all redundant evaluations over nodes that are not of that attribute name and thus achieving good optimization on execution time.

As for XM5(e), the partitioning point is just after the step of `bidder`, of which there are 597797 children nodes. For example, the first subquery of XM5(e) is `/site/../bidder[position()= 1 to 49816]/increase`, letting P = 12. Note that the number of nodes that matches `/site/open_auctions/open_auction` is not 1 but 120000. In this case, when BaseX evaluates the first subquery, it actually traverses all 120000 `open_auction` nodes and evaluates the first 49816 child nodes `bidder` of each `open_auction`. We investigate the number of children of `open_auction` and the max number is only 62. This means that only the first subquery can retrieve resultant nodes, while the rest nodes simply obtain nothing. From this result, we observe that to utilize position-based query partitioning strategy, we should add to add '[pos]' to the step that can make more parallelized subqueries to have results.

Breakdown of Execution Time

In this section, we investigate execution times in greater details by analysing the breakdown of execution time shown in Table 4.5. All the settings are the same as that of data partitioning.

We first observe that for most queries we can reduce execution time by adding more threads. For example, XM1(d), XM3(d), XM4(d) and XM5(d) can be apparently accelerated. While for XM2(d), and XM5(e), the execution times are actually increased. Besides the reason given in the previous section, there is another reason introduced in Section 4.5.2 that the parallelization also brings overhead compared to the original query and it increases with respect to the number of threads increased. We also notice that XM6(d) is improved very limited. This is because the imbalance of the input XML document `xmark10.xml`, i.e. the six children of the root contain quite different amount of descendant nodes, thus making the reduction of execution time by adding threads not very obvious.

We first observe that for most queries we can reduce execution time by adding more threads. For example, XM1(d), XM3(d), XM4(d) and XM5(d) can be apparently accelerated. While for XM2(d), and XM5(e), the execution times are actually increased. Besides the reason given in the previous section, there is another reason introduced in Section 4.5.2 that the parallelization also brings overhead compared to the original query and it increases with respect to the number of threads increased. Let us take XM3(d) as an example. When number of threads is doubled from 1 to 2, the speedup is 540.14/327.03=1.65; when 3 to 6, the speedup is 241.33/159.63=1.51; when 6 to 12, the speedup is 159.63/141.82=1.13. We also notice that XM6(d) is improved very limited. This is because the imbalance of the input XML document `xmark10.xml`, i.e. the six children of the root contain quite different amount of descendant nodes, thus making the reduction of execution time by adding threads not very obvious.

4.6 Observations and Perspectives

In this section, we have revived data partitioning [21] on top of BaseX and experimentally demonstrated, in the best case of non-trivial queries, 4-fold speedup on a 12-core server. We discuss possible improvement of BaseX in terms of partitioning strategies and further perspectives on BaseX.

4.6.1 BaseX Extensions Desirable

Since the implementations of query partitioning is relatively simple and mainly influenced by the structure of input XML documents as we observed, we mainly focus on the implementations of data partitioning strategies in this discussion.

Although the client-side implementation generally fell behind the server-side one in our experiments, it has an advantage that it requires less functionality of XML database servers. If that performance gap were filled, the client-side one would be preferable. Since the performance gaps in prefix queries were small even when their result sizes were more than 1 MB, the difference of cost between sending prefix results to (local) clients and storing them in a server was marginal. The dominant factor was the cost of sending prefix results back in the form of long suffix queries and parsing them on a server. A promising approach to reducing this overhead is client-server streaming of the starting nodes of a suffix query. Since one suffix query is in common applied to many starting nodes, if the suffix query is given to a server in advance, the server can apply it successively to incoming starting nodes and stream out results to clients. With this streaming functionality additionally, the client-side implementation would perform fast nearly to the server-side one.

There is also room for improvement of the server-side implementation. We store block-partitioned arrays into an in-memory database as text parts and then deserialize them to sequences. This is, to the best of our knowledge, the most efficient way of preserving arrays on top of the current BaseX server, but is merely a workaround because its serialization/deserialization is redundant. The most efficient way is obviously to keep XQuery data structures as they are on top of a server. We consider that

it would not necessitate a drastic change of BaseX. Only demand-driven serialization and new function `deserialize` suffice for it as follows. When XQuery values are put into a text part of an in-memory database, they are not serialized immediately but keep their representations. They will be serialized for a query just before the query tries to read the text part. If a query applies `deserialize` to the text part, it returns their original representations in zero cost. It is worth noting that because in-memory databases in BaseX will never be stored on disks, demand-driven serialization per se is worth implementing to avoid serialization into text parts not to be accessed.

4.6.2 Further Perspectives

Both data partitioning and index-based optimizations have worked together well only with the simple way described in Section 4.4. Both lie in the same spectrum in the sense that performance gain is contingent on the statistics of a given document. In fact, document statistics are known to be useful for data partitioning (as well as query partitioning) [20]. Statistics maintained together with indices in BaseX should therefore be utilized for data partitioning together with index-based optimizations. If we implement data partitioning into BaseX, a close integration with both would be naturally feasible. Besides, statistics will be available even outside BaseX by using *probing* queries that count nodes hitting at a specific step. The cost of several probing queries in advance to data partitioning would matter little because simple counting queries are quite fast on BaseX. By using node counts, we can avoid the situation of an insufficient number of prefix query results found in XM1(a). It will be a lightweight choice in the sense of preserving a black-box use of BaseX.

It is challenging yet promising to extend data partitioning to distributed databases with BaseX. The top part of a document to be traversed by prefix queries can be either centralized or replicated. Bottom parts to be traversed by suffix queries should be distributed as separate XML databases. Because the whole database forms a simple collection of XML documents, horizontal fragmentation [59] will be well-suited but it can incur imbalance in size among fragments. Balanced-size cheap fragmentation based on partial trees [49] will be promising for the complement to it. Existing work [37] on

querying large integrated data will be relevant. Hybrid partitioning [21], which is combination of data partitioning and query partitioning, would become important because query partitioning requires synchronization only in merging partial results and the number of synchronizations matters more in distributed databases. Fragmentation-aware hybrid partitioning is worth investigating. The most challenging part is to implement integration with existing index-based optimizations so as to take account of global information, where our idea described in Section 4.4 will be useful but would not be trivially applicable.

4.7 Summary

In this chapter, we first reviewed existing partitioning strategies in study [21]. Due to the out-of-date problem in both hardware and software of the original study, we studied it in a modern XML processing engine BaseX. Exploiting the features of BaseX, we proposed our implementations for the strategies. We conducted experiment to evaluate our implementations. Based on the experiment results, we draw the following three conclusions.

First, as concluded in [21], it is not obvious if data or query partitioning would be beneficial. In our implementations, these are caused by different factors. For the implementations of data partitioning, we need to process a prefix query before processing the suffix queries in parallel. This is a extra cost compared to the original query and the amout of extra time for processing a prefix query relates to the result size of the prefix query. In case the result size is large, it takes a lot time, such as DBLP1(a) and DBLP2(a). While for the implementations of query partitioning, the imbalance of XML documents have a dramatic influence on the query performance and the speedup.

Second, BaseX optimizer plays an important role in the reduction of execution time. In case when BaseX optimizer is available, execution time can be greatly reduced. A very important feature of the optimizer is that it can also be applied in parallel evaluation. Therefore, it is worth taking the optimizations to reduce the execute time as long as the partition of sub-queries can meet the conditions of the BaseX optimizer.

Third, the experiment results clearly show that we can achieve speedup up to 4, which have strongly proved that the partitioning strategies are available not only on XML trees but also on XML database engines. However, the availability of these strategies is significantly dependent on the implementations and the XML database engine/processor. Properly combined with the features of the XML database engine/processor used for implementation, we can achieve significant performance improvements over the original strategies, such as the server-side implementation of data partitioning strategy.

Chapter 5

Partial Tree: A Novel Tree Structure for Parallel XML Processing

To processing XML in parallel, one basic requirement is to use a way to represent an XML document that is to be processed. When the XML document is small, it is very easy to use common tree structures to represent it. However, when the size is very large, especially when it excesses the memory of a target computer, it becomes difficult to process it. When we intent to process an XML document in a distributed-memory environment, it is rather crucial to use efficient data structure for it in terms of memory consumption and the distributed processing.

In this chapter, we will introduce a novel tree structure for parallel XML processing , namely *partial tree*. Firstly, we introduce partial tree (this part was published [49]). Then, we proposed an efficient index scheme for representing partial tree. We conduct experiments and report the results. Lastly, we summarize this chapter.

5.1 Definitions

In this section, we introduce our novel tree structure, partial tree. partial tree is designed to be used for processing XPath queries over XML documents in parallel. The application of partial tree has two steps. Firstly, we split an XML document into many

chunks. Then, from each of these chunks, we can construct a partial trees by using a partial tree computing algorithm. To deeply understand what partial tree is, we now give some definitions of partial tree.

To begin with, we give the definition of node types in a partial tree. A partial tree contains four different types of element nodes categorized by the based on existance of tags as shown in Figure 5.1. The four types of element nodes are: *close node*, *left-open node*, *right-open node* and *pre-node*. A closed node is a regular node that has both its start tag and end tag. A left-open node has only its start tag, while a right-open node has only its end tag. In case a node is both a left-open node and a right-open node, we call it *pre-node*. For a node that misses any of its tags, we call it *open node*, including left-open node, right-open node and pre-open node.

Open nodes are not a new concept. Kakehi et al [55] proposed an approach for parallel reductions on trees and explicitly used 'open' nodes in their paper. Choi et al. [33] proposed an idea to label nodes in split chunks of a large XML document. Although they did not use the term and also did not form a tree in their study, the idea is similar as that of open nodes.

A pre-open node is a novel idea proposed in our study and is different from the other two open nodes that it comes from no tag, i.e. we do not create a pre-node directly from the corresponding tags. This is because the corresponding tags of a pre-node do not exist in the chunk. This is, on the contrary, the most significant idea of this study, so that we can use it to represent the missing nodes on the path from the root of the whole tree to current chunk.

We also have to understand that a chunk is a part of an XML document and it contains a sequence of tags (the start tags may have attributes) and content nodes. For simplicity, we omit attribute and content nodes in our discussion. From a chunk, we can construct a list of subtrees with open nodes. But we cannot make a partail tree, because the structural information of the whole tree is missing. We need a way to restore the original structure, more specifically to link the current list of subtrees to the root of the original tree(The related part will be introduced in section 5.3). Please also note that (1) since our research focuses on evaluating queries on partial trees, more

specifically, mainly on element nodes, in case no tag is contained in a chunk, we merge it to the next chunk until there is at least a tag existed in the chunk (this is very rare case only when a context is too large or the chunk is too small); Note that (2) all open nodes, including pre-node, are element nodes.

For representing open nodes separately from the close nodes in figures and tables, we add one black dot: • to an open node, representing on which side the node is open. For the example, given a pair of tags, `<A></A>`, we can create an right-open node A• from `<A>` and a left-open node •A from `</A>`. Due to the fact that these two nodes will be on different partial trees, we use *range* to record this information. A range is the pair of two numbers denoting the first and the last ordinals of partial trees where the corresponding pair of tags are located. For example, we split an XML document in 10 chunks and thus we can construct 10 partial trees accordingly. A pair of tags in the original document is split. Its start tag locates on the 2nd partial tree and the end tag on 5th partial tree. Then the rang of this node is [2, 5]. According to the above definitions, a partial tree is defined as a tree structure with open nodes and ranges to represents a chunk of an XML document and can be used for parallel XML processing.

For parallel processing queries, we have the following definitions. Firstly, we add a virtual node VNi into a resultant list for each partial tree, which has the root of PTi as its single child. For example, VN_0 has only a single child $\mathtt{A}_0\bullet$ of pt_0 and is put into the resultant list before the evaluation on pt_0 starts. The evaluation of a step generates a new resultant list of nodes. After the evaluation, we replace the current resultant list with the new list as the results and will be used as the input for the next step. Each node has a *type* denoting its node type, including closed, left-open, right-open and pre-node, and *depth* denoting the number of edges from the node to the root. A node has four member variables to the related nodes: the *parent* to its parent and the *children* to its children. For accessing siblings, it has the *presib* and the *folsib* that points to its preceding-sibling node and following-sibling node, respectively. For distinguishing nodes, we give each node a unique id called *uid*.

Besides the partial tree node, there is also a requirement that we need to know from which partial tree a node comes in distributed memory environments; therefore,

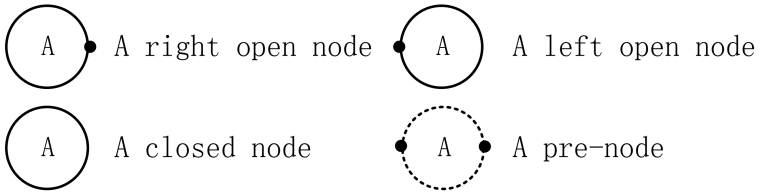


Figure 5.1: Four types of element nodes

we number each partial tree with a unique id denoted as *partial tree id* or shortly *ptid* for distinguishing partial trees. We number *ptid* from 0 to *P-1* (where P is the total number of partial trees) in the document order. For locating any node on a partial tree, we define data type *Link* that holds a ptid and a 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 access any node in constant time.

5.2 Characteristics of Partial Tree

Now we discuss the Characteristics of partial trees. Since the left-open nodes, right-open nodes and pre-nodes are most significant concept in partial trees, we first focus on the properties of these open node.

5.2.1 Properties of Open Nodes

We introduce three properties of open nodes. 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 relationship of open nodes among their siblings.

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 at most one of them has left-open/closed/right-open nodes as its child.*

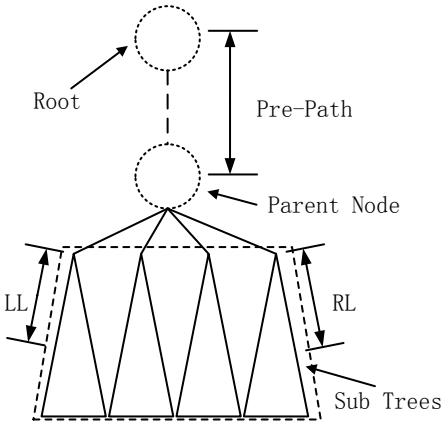


Figure 5.2: The standard structure of partial tree(LL=a list of left open nodes; RL=a list of right-open nodes).

5.2.2 Standard Structure

According the properties of partial trees, the standard structure of partial tree can be described as in Figure 5.2.

A partial tree consists vertically of two parts. The bottom part is a forest of subtrees and the top part is a list of pre-open nodes denoting the path from the root to the bottom part. 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.

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).

5.3 Construction of Partial Trees

Since the structure of partial tree is quite different from ordinary XML trees in terms of open nodes, especially the pre-path. Thus, ordinary XML parsing algorithms,

in turn, do not work for the construction of partial tree. The difference mainly lies in two aspects. Firstly, an XML document can generate only a single XML tree; while in case of partial tree, since a partial tree is constructed from a chunk, the number of partial trees could be many. Secondly, constructing a partial tree is different from parsing an ordinary XML tree, because the pre-path of a partial tree is missing in the corresponding chunk.

For constructing the pre-path, it is important that a partial tree that corresponds to a chunk should satisfy the following three conditions:

(1) the subgraph is connected;

(2) each node parsing from the chunk is in the subgraph, and

(3) the root of the original XML tree is in the subgraph.

For an intuitive grasp, we use the following XML document as the running example.

```
<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 construct an XML tree as shown in Figure 5.3. We number all the nodes of the tree in a prefix order for identification.

To construct partial trees, we first split it 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>`

We can construct partial trees by using these chunks as shown in Figure 5.4.

In the following section, we introduce our partial tree construction algorithm with the example. Our algorithm consists of three phases. In the first phase, we construct multiple lists of subtrees that have some open nodes from parsing chunks of the input XML document. Second, we compute pre-path for each list of subtrees with all the

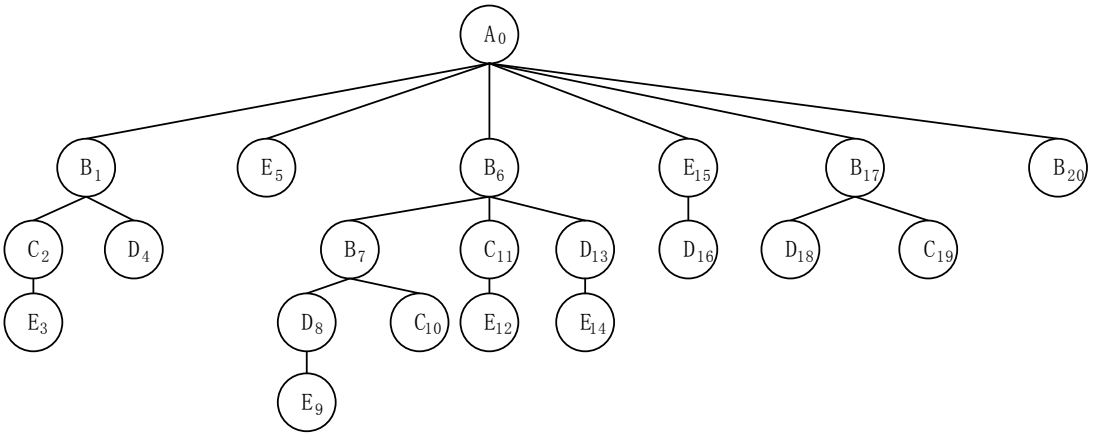


Figure 5.3: an XML tree from the given XML string

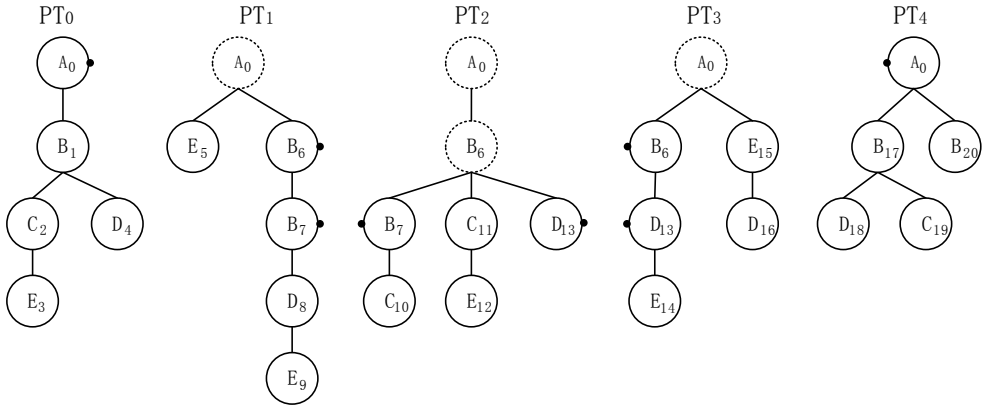


Figure 5.4: Partial trees from the given XML string.

open nodes. Last, we add pre-paths to the corresponding list of subtrees to complete the partial tree construction. We will give introduction to the three-phase construction algorithm of partial trees. Furthermore, for designing query algorithms, we also use the statistics information of open nodes, i.e. ranges of open nodes. With such information, we can easily access open nodes and synchronize the query results among partial trees.

5.3.1 Construction of Subtrees From Parsing XML Chunks

We design an algorithm that parses the input XML string into a similar tree by using an iterative function with a stack, which is similar to ordinary XML parsing algorithms.

Firstly, we deal with nodes with missing tags from input chunks. During parsing,

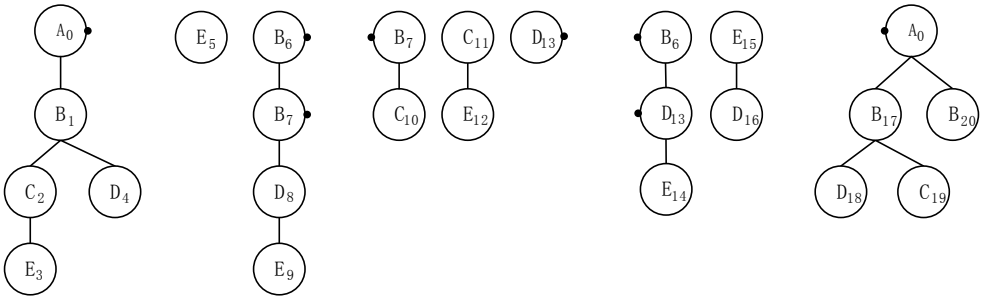


Figure 5.5: Subtrees from parsing chunks

we push start tags onto the stack. When we meet an end tag, we pop a last start 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 tag (either start tag or end tag) 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 into a single one as it is, 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 a single chunk. For example, we can construct nine subtrees from parsing the five chunks above as shown in Figure 5.5. 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.

5.3.2 Pre-path Computation

The key idea of computing the pre-path for each partial tree is to make use of open nodes. This is because the missing parent and ancestor nodes are caused by splitting the original pairs of tags of those nodes. We need the information for creating the pre-paths.

The algorithm GetPrepath outlines the computing steps for pre-path. Since one chunk may generate more than one subtree, the input is a list of lists of subtrees. The length of the list is equal to the number of partial trees, i.e. one chunk makes one list

Algorithm GetPrepath(STS)
Input: STS: a list of subtree lists
Output: an list of partial trees
1: /* open nodes in LLS or RLS are arranged in top-bottom order */
2: **for all** $p \in [0, P)$ **do**
3: $LLS_{[p]} \leftarrow SelectLeftOpenNodes(STS_{[p]})$
4: $RLS_{[p]} \leftarrow SelectRightOpenNodes(STS_{[p]})$
5: /* Prepath-computation and collecting matching nodes */
6: $AuxList \leftarrow []$
7: **for** $p \in [0, P-1)$ **do**
8: $AuxList.AppendToHead(RLS_{[p]})$
9: $AuxList.RemoveLast(LLS_{[p+1]}.Size())$
10: $PPS_{[p+1]} \leftarrow AuxList$
11: /* Add pre-nodes to subtrees */
12: $PTS \leftarrow []$
13: **for** $p \in [0, P)$ **do**
14: **for** $i \in [0, PPS_{[p]}.Size() - 1)$ **do**
15: $PPS_{[p][i]}.children$.Add($PPS_{[p][i+1]}$)
16: $PPS_{[p]}.last.children$.Add($STS_{[p]}$)
17: $PTS_{[p]} \leftarrow PPS_{[p][0]}$
18: **return** PTS

of subtree, and is represented as P.

The algorithm GetPrepath has three phases. In the first phase, it selects all the left-open nodes into LLS and all the right-open nodes into RLS (line 2–4). $LLS_{[P]}$ collects the left-open nodes of the P-th partial tree, likewise we have $RLS_{[P]}$. Note that the nodes in $LLS_{[P]}$ or $RLS_{[P]}$ are arranged in order from the root to the leaves. For example, in Table 5.1, we select all the open nodes and add them to corresponding lists.

In the second phase, we perform the pre-path computation. Once we split an XML document into two halves, there are two partial trees can be constructed from the splitting and they have the same number of open nodes on the splitting side. Given two consecutive partial trees, the number of the right-open nodes (including pre-nodes) of the left partial tree is the same as the number of the left-open nodes of the right partial tree (including pre-nodes as well). This is a very important feature and we exploit it for computing the pre-paths of partial trees.

Table 5.1: 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]	[]

Table 5.2: Results of pre-path computation in AUX(an auxiliary list for storing pre-nodes)

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

In the algorithm, we first add the pth RLS to the head of an auxiliary list $AuxList$ (line 8), and then we remove the same number of nodes as the number of $(p-1)$th LLS (line 9). Last, we keep the nodes in the $AuxList$ to the $(p+1)$th PPS, which holds the pre-nodes for each partial tree. Table 5.2 shows the results of pre-path computation for the given example.

In last phase, we add the resultant pre-nodes to the corresponding partial trees and copy the nodes from $PPS_{[p]}$ to $PTS_{[p]}$ as the results for output. Because the pre-nodes in the pre-path are also open nodes, we list all open nodes for each partial trees in Table 5.3. Then, the pre-path computation is complete. For the given example, we obtain the partial trees as shown in Fig. 5.4.

The essence of algorithm is to utilize the open nodes of all subtrees to compute the missing structural information and depth the each subtree so that we can figure out the relationship of these subtrees parsed from each chunk to the root of the original XML tree. Eventually, we obtained the pre-path for each chunk and construct partial trees.

Table 5.3: 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]	[]

Table 5.4: 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)]	[]

5.3.3 Creation of Ranges of Open Nodes

Once an XML node is split, it generates two or more open nodes of the same node in consecutive partial trees. For example, as we can see in Figure 5.4, $\texttt{B}_6\bullet$ on pt_1, $\bullet\texttt{B}_6\bullet$ on pt_2, and $\bullet\texttt{B}_6$ on pt_3 are created from the same node $\texttt{B}_6$. For locating the open nodes of the same node, which are distributed in different partial trees, we use two integers *start* and *end* for an open node. With these two integers, we can know the partial trees that have matching nodes of the same open node. Note that after adding pre-nodes to form a partial tree, the open nodes from the same node also have the same depth. Therefore, we can locate all the open nodes by using these ranges. After computation, we obtain the ranges of nodes as shown in Table 5.4.

By using these ranges, we can easily locate the partial trees for the matching nodes of the same node. For example, the range of A_0 is (0, 4), that means we can locate the same nodes of $\texttt{A}_0$ from pt_0 to pt_4. As we can see, there are $\texttt{A}_0\bullet$, $\bullet\texttt{A}_0\bullet$, $\bullet\texttt{A}_0\bullet$, $\bullet\texttt{A}_0\bullet$, and $\texttt{A}_0\bullet$ on pt_0 to pt_4, respectively.

5.4 Eavalute XPath Queries on Partial Trees

A significant advantage of using partial tree is that it makes the parallelization of evaluation of XPath queries possible by simply evaluating the same query on partail trees separately. However, we also need to notice that it also brings challenges in designing query algorithms on partial trees. Generally speaking, there are three main difficulties as follows.

First, since partial trees are created from chunks of an XML document, a node may be separated and thus lie in different partial trees. This leads to a possible situation that multiple open nodes may stem from the same node and distributed in different partial trees as discussed in Section 5.3.3. Thus, in case when one of such open nodes is selected in a partial tree (e.g., $\texttt{B}_6\bullet$ on pt_1), the other corresponding nodes ($\bullet\texttt{B}_6\bullet$ on pt_2 and $\bullet\texttt{B}_6$ on pt_3) should also be selected for consistency.

Second, although partial trees have all the parent-child edges of their nodes, the sibling-relation that is split among partial trees may be missing. For example, $\texttt{B}_1$ has five following-sibling nodes in the original XML tree, but in pt_1, there is no following-sibling nodes of $\texttt{B}_1$ because of splitting. When we perform queries with `following`, `following-sibling`, `preceding` or `preceding-sibling`, the results may be in another (possibly far) partial tree. We 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 will show this in the following part of this section). We also need an algorithm to propagate the information over partial trees for queries with predicates.

In this section, we develop algorithms for evaluating XPath queries on a set of partial trees. We first show the outline of the algorithms and then describe the details of how the query algorithms work for evaluating XPath 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`

After the introduction of the algorithms, we also discuss the complexity of our algorithms at the end of this section.

5.4.1 Overall Query Algorithm

When processing an XPath query, we evaluate it step by step in order. For storing the results of a step, we define a resultant list of nodes for each partial tree i.e. the are P resultant lists given P partial trees. The evaluation of a step is applies to each node in the resultant list.

Algorithm 1 outlines the big picture of our XPath query algorithms. The input includes an XPath query to be evaluated and a set of partial trees generated from an XML document. The output is a set of resultant nodes that matchesthe XPath query, each of which is associated with a corresponding partial tree.

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

As we know, an XPath query consists of one or more location steps, and in our algorithm they are processed one by one in order. For each step, Algorithm 1 calls the corresponding sub-algorithm based on the axis of the step and updates the intermediate results (line 4) in the resultant lists for each turn. Lines 6–9 will be executed in case the input XPath query has a predicates. We will explain this part later in Section 5.4.3.

5.4.2 Queries without Predicate

Downwards Axes

Algorithm 2 shows the procedure for evaluating a step with a child axis. The input $InputList_{[P]}$ is a list of lists of nodes and is the results of evaluating the previous location

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]
```

Figure 5.6: Overall algorithm of XPath query for partial trees

step on every 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 shows the procedure for evaluating a step with a descendant axis. Starting from every node in the input, it traverses partial trees by depth-first search along with a stack. To avoid redundant traversals on the same node, we add the *isChecked* flag for each node (lines 8–9) so that we can evaluate each node only once. The function SetIsChecked(pt_p, *false*) is to reset all the *isChecked* of nodes in pt_p. Note that we can reduce the worst-case complexity by using this flag from square to linear with respect to the number of nodes.

Now let us look at our running example Q1. The whole evalution of Q1 is listed in Table 5.5. For the first step `child::A`, since VN_i has only one child that is the root of the ith partial tree, we obtain it as the result for each partial tree as shown in the second row of the table. Then we perform the next step `descendant::B` independently for each partial tree from the result of the first step. We evaluate the step `descendant::B` on each of the nodes in the resultant list of the previous step. For example, for $\mathtt{A}_1\bullet$ on pt_1, there is only a node $\mathtt{B}_1$ that matches the query and is then selected and put into a new list. The results of this step for each partial tree are listed in the fourth row of Table 5.5. For the third step `descendant::C`, the algorithm works in a similar way.

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:     ResultList_[P] ← QUERY⟨step.axis⟩(pt_[P], ResultList_[P], step.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]
```

Figure 5.7: Query algorithm for downwards axes

The results up to `descendant::C` are lised in the last row of Table 5.5. It is worth noting that the *isChecked* flag now works. For example, on pt_1, starting from $\texttt{B}_6\bullet$, we traverse $\texttt{B}_7\bullet$, $\texttt{D}_8$, $\texttt{E}_9$, and then starting from $\texttt{B}_7\bullet$, we can stop the traversal immediately.

Upwards Axes

In querying of a step with downward axes, the algorithms have nothing different to partial trees compared to ordinary XML trees. This is because closed nodes can be processed normally and open nodes are due to Property 1 in Section 5.2. Let an

Table 5.5: Evaluating downward steps of Q1

Process	pt_0	pt_1	pt_2	pt_3	pt_4
Input	$[VN_0]$	$[VN_1]$	$[VN_2]$	$[VN_3]$	$[VN_4]$
`child::A`	$[A_0\bullet]$	$[\bullet A_0\bullet]$	$[\bullet A_0\bullet]$	$[\bullet A_0\bullet]$	$[\bullet A_0]$
`descendant::B`	$[B_1]$	$[B_6\bullet, B_7\bullet]$	$[\bullet B_6\bullet, \bullet B_7]$	$[\bullet B_6]$	$[B_{17}, B_{20}]$
`descendant::C`	$[C_2]$	$[\,]$	$[C_{10}, C_{11}]$	$[\,]$	$[C_{19}]$

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.

However, this discussion does not hold for the queries with upwards axes. In such case 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 for sharing the selected nodes when we process the upwards axes.

Algorithm 4 shows the procedure for evaluating a step with a 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.

The ShareNodes function consists of two phases. In our implementation, there are two phases of communication. We first send open nodes of all partial trees to a process and then send back the necessary data to the according partial trees. In between these two phases, we compute for each open node about which partial trees should be shared with it.

In the first phase, it collects all the selected open nodes from all 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. In the second phase, after the call of ShareNodes function, all the open nodes that are from the same node are selected in corresponding partial trees.

Now, let us continue the running example Q1 for its last step as shown in Table 5.6.

Table 5.6: Evaluating the last step of Q1

Process	pt_0	pt_1	pt_2	pt_3	pt_4
Input	$[C_2]$	$[]$	$[C_{10}, C_{11}]$	$[]$	$[C_{19}]$
`parent::B`	$[B_1]$	$[]$	$[\bullet B_7, \bullet B_6 \bullet]$	$[]$	$[B_{17}]$
ShareNode	$[B_1]$	$[B_6 \bullet, B_7 \bullet]$	$[\bullet B_7, \bullet B_6 \bullet]$	$[\bullet B_6]$	$[B_{17}]$

For the `parent::B` after the `descendant::C`, we first directly select the parent nodes of the intermediate results from the results of the previous step independently. For the running example, B_1 is selected since it is the parent of C_2 on pt_0, while for $\bullet B_7$ and $\bullet B_6 \bullet$, they are the parents of C_{10} and C_{11} respectively. The results are listed in the second row of Table 5.6.

Here, unfortunately the node $\bullet B_7$ is selected on pt_2, but its corresponding node on pt_1, i.e. $B_7 \bullet$, has not been selected yet. We then call the ShareNodes function. By collecting all the open nodes from all the partial trees, we have the list $[\bullet B_7, \bullet B_6 \bullet]$. Since they have ranges $(1, 2)$ and $(1, 3)$, respectively, pt_1 receives two nodes $B_7 \bullet$ and $B_6 \bullet$, pt_2 receives two nodes $\bullet B_7$ and $\bullet B_6 \bullet$, and pt_3 receives one node $\bullet B_6$. By taking the union with the previous intermediate results, we obtain the final results as shown in the last row of Table 5.6.

Now, after evaluating the last step of Q1, the evaluation of the whole query Q1 is complete. The resultant lists in the last row of Table 5.6 from evaluating the last step is then the final results of Q1.

Intra-sibling Axes

The `following-sibling` or `preceding-sibling` axes retrieve nodes from a set of sibling nodes of an intermediate node. In our partial trees, a set of those sibling nodes might be divided into two or more partial trees. Therefore, these 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, since preceding-sibling is only different in a opposite direction compared to following-sibling axis. Al-

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 ShareNodes(pt_[P], OutputList_[P])
```

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

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]
```

Figure 5.8: Query algorithms for upwards axes

gorithm 6 shows the procedure for evaluating a step with a following-sibling axis, which consists of four phases: local query, preparation of open nodes, regrouping open nodes, 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. This operation is to be used for evaluating sibling nodes on the following partial trees. The latter two conditions (lines 14, 15) are rather easy: we will ask a remote query if the parent node

Table 5.7: Evaluating the location steps of Q2

Process	pt_0	pt_1	pt_2	pt_3	pt_4
Input	$[VN_0]$	$[VN_1]$	$[VN_2]$	$[VN_3]$	$[VN_4]$
`descendant::B`	$[B_1]$	$[B_6\bullet, B_7\bullet]$	$[\bullet B_6\bullet, \bullet B_7]$	$[\bullet B_6]$	$[B_{17}, B_{20}]$
`following-sibling::B`	[]	[]	[]	[]	$[B_{20}]$
remote `Parent::A`	[]	$[\bullet A_0\bullet]$	$[\bullet A_0\bullet]$	$[\bullet A_0\bullet, \bullet B_6]$	$[\bullet A_0]$
remote `child::B`	[]	$[B_6\bullet]$	$[\bullet B_6\bullet]$	$[\bullet B_6]$	$[B_{17}, B_{20}]$

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 itself is right open. Notice that if a node is right open then it should have a corresponding left-open node in 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 the evaluation of `descendant::B` , we have the intermediate results in the third row of Table 5.7. In the first phase of `following-sibling::B`, we get the results in the fourth row. Then, we collect the parent nodes that satisfies the conditions (lines 13–15). Such nodes and their ranges are: $A_0\bullet$ with range [1,4] (on pt_0), $\bullet B_6\bullet$ with range [3,3] (on pt_2), and $\bullet A_0\bullet$ 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 in the fourth row of the table. Starting from these intermediate results, we select their children and obtain the results as shown in the last row of Table 5.7. 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.

5.4.3 Queries with Predicate

Predicates in this study are filters that check the existence of matched nodes by steps that have no predicates. Our algorithm for handling predicates consists of three phases: 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 that have no predicates. In the

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

Figure 5.9: Algorithm for Following-sibling axis

querying in predicates, we also attach a link to each of 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 $x \rightarrow \{\mathrm{pt}_i.y\}$ in the running example, both of which mean node x is selected and it has a link to node y on pt_i.

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*

Figure 5.10: Query algorithm for handling predicate

Table 5.8: Prepare the predicate of Q3

Process	pt_0	pt_1	pt_2	pt_3	pt_4
Input	$[VN_0]$	$[VN_1]$	$[VN_2]$	$[VN_3]$	$[VN_4]$
`descendant::B`	$[B_1]$	$[B_6\bullet, B_7\bullet]$	$[\bullet B_6\bullet, \bullet B_7]$	$[\bullet B_6]$	$[B_{17}, B_{20}]$
Prepare predicate	$[B_1 \rightarrow \{pt_0.B_1\}]$	$[B_6\bullet \rightarrow \{pt_1.B_6\bullet\}, B_7\bullet \rightarrow \{pt_1.B_7\bullet\}]$	$[\bullet B_6\bullet \rightarrow \{pt_2.\bullet B_6\bullet\}, \bullet B_7 \rightarrow \{pt_2.\bullet B_7\}]$	$[\bullet B_6 \rightarrow \{pt_3.\bullet B_6\}]$	$[B_{17} \rightarrow \{pt_4.B_{17}\}, B_{20} \rightarrow \{pt_4.B_{20}\}]$

Preparing Predicate

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

For example in Q3, after evaluating `descendant::B`, we have the resultant lists before the predicate evaluation as shown in the third row of Table 5.8. Then, by calling PREPAREPREDICATE, we have the intermediate results as shown in the last row of the table. Note that (1) all the links point to the nodes themselves at the beginning and (2) the resultant lists in the last row of the table are newly created apart from the current resultant list.

Evaluation of Steps within A Predicate

The evaluation of inner steps of a predicate is almost the same as that without predicate. Algorithm 9 shows the procedure for evaluatinga step with a child axis in the predicate; the key differences are the type of intermediate values and the duplication of links.

There is another important difference for the descendant, ancestor, following-sibling, and preceding-sibling. In the querying without predicate, we used the *isChecked* flag to

Table 5.9: Evaluate inner steps of the predicate in Q3

Process	pt_0	pt_1	pt_2	pt_3	pt_4
Input	$[B_1 \rightarrow \{pt_0.B_1\}]$	$[B_6\bullet \rightarrow \{pt_1.B_6\bullet\}, B_7\bullet \rightarrow \{pt_1.B_7\bullet\}]$	$[\bullet B_6\bullet \rightarrow \{pt_2.\bullet B_6\bullet\}, \bullet B_7 \rightarrow \{pt_2.\bullet B_7\}]$	$[\bullet B_6 \rightarrow \{pt_3.\bullet B_6\}]$	$[B_{17} \rightarrow \{pt_4.\bullet B_{17}\}, B_{20} \rightarrow \{pt_4.B_{20}\}]$
local `following-sibling::B`	[]	[]	[]	[]	$[B_{20} \rightarrow \{pt_4.\bullet B_{17}\}]$
remote queries	[]	$[B_6\bullet \rightarrow \{pt_0.B_1\}]$	$[\bullet B_6\bullet \rightarrow \{pt_0.B_1\}]$	$[\bullet B_6 \rightarrow \{pt_0.B_1\}]$	$[B_{17} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}$ $B_{20} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}]$
merge link	[]	$[B_6\bullet \rightarrow \{pt_0.B_1\}]$	$[\bullet B_6\bullet \rightarrow \{pt_0.B_1\}]$	$[\bullet B_6 \rightarrow \{pt_0.B_1\}]$	$[B_{17} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}, B_{20} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6, pt_4.\bullet B_{17}\}]$
`child::C`	[]	[]	$[\bullet C_{11} \rightarrow \{pt_0.B_1\}]$	[]	$[C_{19} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}]$

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 for the evaluation of the inner steps of the predicate as shown in Table 5.9. 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 only different is the nodes to be processed, which have links with them. We obtain the results as shown in the third row of the table.

The remote queries are different from steps that are not in a predicate. Although selected nodes are the same as before, they may have multiple links and are stored in newly created lists. For example, B_{17} and B_{20} in pt_4 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 as shown in the fourth row of the table.

For example, let us consider B_{20} in pt_4. The local result of it is $B_{20} \rightarrow \{pt_4.B_{20}\}$, and the remote results is $B_{20} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}$. After merging link, we have $B_{20} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6, pt_4.B_{20}\}$ as the result.

Similarly, by applying the following step `child::C`, the intermediate results are shown in the last row of Table 5.9. Note that in pt_4, the resultant node $[C_{19} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}]$ is a child of B_{17}. Thus, it follows the link of B_{17}.

Table 5.10: Process the predicate in Q3

Process	pt_0	pt_1	pt_2	pt_3	pt_4
Input	[]	[]	$[\bullet C_{11} \rightarrow \{pt_0.B_1\}]$	[]	$[C_{19} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}]$
Process predicate	$[B_1]$	[]	[]	$[\bullet B_6]$	[]
ShareNode	$[B_1]$	$[B_6\bullet]$	$[\bullet B_6\bullet]$	$[\bullet B_6]$	[]
`child::C`	$[C_2]$	[]	$[C_{11}]$	[]	[]

Processing Predicate

Finally, we process the intermediate results to obtain the results after filtering of predicate. Algorithm 8 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, the results are shown in the third row of Table 5.10. A link $C_{11} \rightarrow \{pt_0.B_1\}$ in the intermediate result of pt_2 adds node B_1 to the result list of pt_0 and $C_{19} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}$ in the intermediate results of pt_4 adds two nodes, B_1 on pt_0 and $\bullet B_6$ on pt_3, respectively. We then apply the ShareNodes function and obtain the intermediate results as in the second last row of Table 5.10.

The last step simply calls the processing of the step with a child axis, and the final results for Q3 are in the last row of the table. Then, the query of Q3 is complete. All the nodes in the resultant lists are the final results.

5.4.4 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.

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

Input: $pt_{[P]}$: an indexed set of partial trees
$InputList_{[P]}$: 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 ← [ ]
3: for i ∈ [0, P) do
4:    AllLinks ← AllLinks ∪ [(p', i')|(n', (p', i')) ∈ InputList_p]
5: for i ∈ [0, P) do
6:    Activated_p ← [n | (p', i') ∈ AllLinks, p = p', n.uid = i']
7: return ShareNodes(pt_[P], Activated_[P])
```

Figure 5.11: Query algorithm for handling predicate

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

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

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

Figure 5.12: Query algorithm for child axis in a predicate

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 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 5.11 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

Table 5.11: 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			$O(P^2K^2)$	$O(P^2K^2)$

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 $O(KN/P + P^2K^2)$.

5.5 BFS-array based implementation

Based on the idea of partial tree, we can divide a large XML document into chunks and distribute the evaluation of XPath queries over partial trees constructed from the chunks of the document. However, without appropriate implementation, it is still difficult to process XML document efficiently in case when they are large. This is because the requirements for processing the large XML documents are more strict, particularly that the memory and random access to tree nodes become more crucial.

In this section, we propose an efficient implementation of partial tree based on two indexes: BFS-array index and grouped index. For better use, we also introduce attribute nodes and values of nodes into our implementation. To begin with, we give an example first. Considering the following XML document that has been divided into four chunks, we can construct four partial trees as shown in Figure 5.13.

chunk$_0$: `<r><b><d><c>txt1</c></d><a at="1"></a></b><b><d>`

chunk$_1$: `<c>txt2</c><d><c>txt3</c></d></d><a at="2"></a><d>`

chunk$_2$: `<c>txt4</c><c>txt5</c></d><a at="3"></a></b><b><d>`

chunk$_3$: `<c>txt6</c></d><d><d><c>txt7</c></d></d></b></r>`

In this XML document, there are three attributes and seven text values. An attributes is denoted as a rectangle boxes with both the attribute name and the attribute value. The Value of a node is denoted as a rectangle box with only the text of the node.

Now, let us consider how partial tree work for this XML document that has four chunks. Since attribute nodes are inside a tag, which will not be separated, and the values of nodes that can be considered as a regular tree node. We then can construct four partial trees as shown in Figure 5.14.

For the four partial trees, we represent element nodes, attribute nodes and value nodes consistent as the original XML tree. Since the partition only affects the element node, from which the open nodes are only generated and attribute nodes and content nodes can be simply implemented by the idea of partial tree. As we have introduced

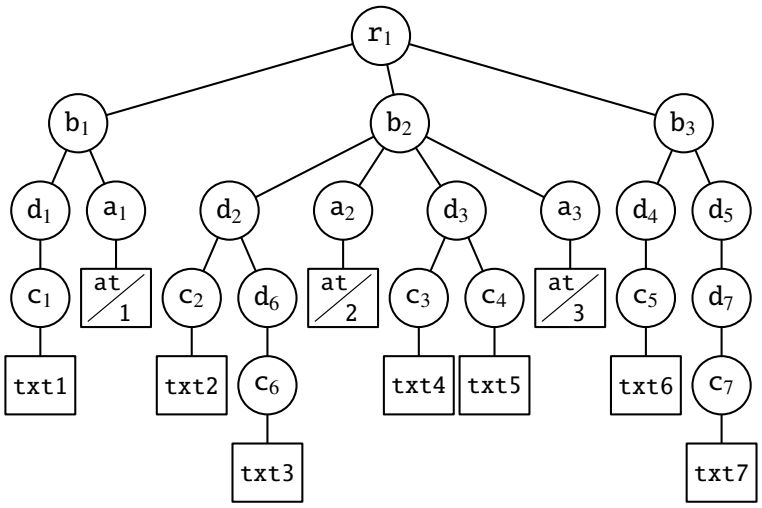


Figure 5.13: An example XML tree with values.

in Section 5.3, the split tags are merged in case when the split position falls inside a tag and thus splits the tag into two halves. In case of a text node is split into two sub texts and separated on different partial trees, we simply merge the split two sub texts into one and leave it on one partial tree. Thus this makes the algorithm consistent.

The partial trees in the previous section provide a nice fragmentation of an XML tree, making it possible for data parallel processing. To develop a high-performance query framework, we still need to design concrete data representation taking the following two issues into consideration.

Expressiveness

Originally indexing (or labeling) was considered to put shredded XML data into databases [18, 81], but its expressiveness is very important to accelerate queries.

Compactness

In the case we repeatedly apply several queries on the same data, we can put all the indexes in memory to avoid expensive I/O cost.

The first design choice is about the updates of XML data. In general purpose framework, efficient support of updates is an important issues and several frameworks support updates with sophisticated indexing such as ORDPATH [81]. However, such an index with the update capability tends to be large and complicated to handle.

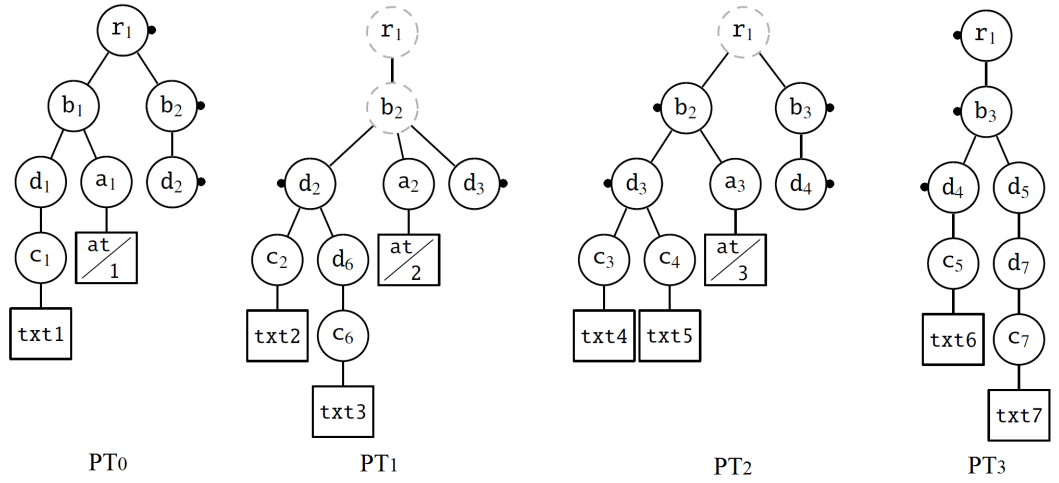


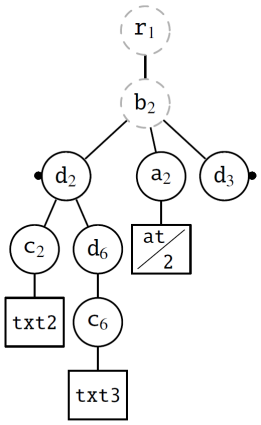
Figure 5.14: Partial trees with values from the XML document.

In this study, we decided not to allow users to update XML data, which makes the implementation much simpler and faster. We expect that users can fulfill their objective without updating the XML data themselves, if we provide some programming interface over the framework.

The second design choice is about the functionality that the indices provide. An important goal of this work is to support queries with not only the `child` and `descendant` axes but also order-aware ones such as `following-sibling` and `following`. To achieve our goal, the following functions should be efficiently implemented.

- Function getChildren(x) returns all the children of node x.
- Function getParent(x) returns the parent of node x.
- Function nextSibling(x) returns the next (right) sibling of node x.
- Function prevSibling(x) returns the previous (left) sibling of node x.
- Function isDescendant(x, y) returns true if node x is a descendant of node y.
- Function isFollowing(x, y) returns true if node x is strictly after node y in the document order.

In tree form



PT1

BFS-array indexing

index	tag	type	st	ed	par	ch
1	5 (r)	N-OO	0	196	0	2
2	2 (b)	N-OO	42	142	1	3
3	4 (d)	N-OC	45	81	2	6
4	1 (a)	N-CC	81	95	2	8
5	4 (d)	N-CO	95	124	2	9
6	3 (c)	N-CC	48	59	3	9
7	4 (d)	N-CC	59	77	3	10
8	6 (at)	A	88	88	4	11
9	0	T	51	55	6	11
10	3 (c)	N-CC	62	73	7	11
11	0	T	65	69	10	12

Grouping Array

tag	index
0	[9, 11]
1	[4]
2	[2]
3	[6]
4	[3, 5, 7]
5	[1]
6	[8]

Figure 5.15: A partial tree in tree form and in index form: BFS-/Grouping Array

- Function $\mathsf{getNodesIn}(t, x)$ returns all the nodes with tag t in the subtree rooted at x.

We design two index sets (Fig. 5.15) to provide these functions keeping the indices compact. A node has the following fields:

- *tag*: tag names (they are short integers that map to the strings),
- *type*: type of nodes including the four node types,
- *st*: start position (the position in the file to avoid global counting), and

- *ed*: end position.

The first index, *BFS-array*, lists all the nodes in the order of the breadth first search (BFS). Every node has two integer pointers to its parent (*par*) and the first child (*ch*) in this list. With the BFS order and these two pointers, we can compute functions getChildren, getParent, nextSibling, and prevSibling efficiently. The second index, *Grouped-array*, groups the nodes by their tag names and then sorts the nodes in the groups by their start position. With this index, we can evaluate the function getNodesIn efficiently.

In our implementation, we used 2 bytes for *tag*, 1 bytes for *type*, 8 bytes for *st*, 8 bytes for *ed*, 4 bytes for *par*, 4 bytes for *ch*, and 4 bytes for *idx*. (Though total file size could exceeds 32 bits, we assume that the number of elements in a single partial tree can fit in 32 bits.) The total size needed for representing a node is 2+1+8+8+4+4+4 = 31 bytes, which is much smaller than several implementation of DOM trees or databases. This is a key to achieve high-performance evaluation of queries.

5.6 Evaluation

In this section, we evaluate the performance of our indexing scheme. We conducted two experiments to evaluate two aspects of the scheme.

The first experiment is to investigate the absolute query time with 100s GB of XML data with up to 32 computers.

The second experiment is to explore the scalability of the indexing scheme for processing a 16.8 GB of XML documents with up to 64 workers on 8 computers. In order to understand the performance of our indexing scheme, we also compare ours with BaseX, the state-of-the-art XML database engine.

5.6.1 Absolute Query Time

We conducted experiments to investigate the best absolute query time in comparison with BaseX in this section.

Hardware Settings

We used were Amazon Elastic Compute Cloud (EC2) M3 for this experiment. M3 Instances are general purpose compute instances that are powered by E5-2670 v2 (Ivy Bridge). It offers a balance of compute, memory, and networking resources for a broad range of workloads. The instance type we used was m3.2xlarge, which is equipped with 30 GB of memory and 2 X 80 GB of SSD, running Amazon Linux AMI 2016.09.0. The network among EC2 instances was a local network and the network speed is 1 gbps. The java running on m3.2xlarge was 64-Bit JVM (build 25.91-b14).

Datasets and XPath Queries

There were three datasets used in this experiment, the statistics of which are shown in Table 5.12. For XMark [94] datasets, we used the XML document generator *xmlgen* from XMark project[1]. The XMark xmlgen takes an float number f to determine the size of generated datasets. For the experiments on a single EC2 instance, we used two XML datsets: DBLP and xmark100 ($f = 100$). For the parallel processing, we used xmark2000($f = 2000$) and UniProtKB. The UniProtKB dataset has a root element with a large number of children with the same tag name and thus can be easily well-formed to be processed by multiple processors. In contrast, XMark datasets whose root has only six children with different tag names, each containing different amounts of data, makes it difficult to be well-formed. Table 5.13 shows 15 queries for three cases: XQ1 and UQ1 to test long queries with nested predicates; XQ2, DQ1, DQ2, UQ2, UQ4 and UQ5 to test backward axes; the rest to test order-aware queries.

Evaluate Queries on a Single EC2 Instance

This experiment is to investigate the query performance on a single EC2 instance. In this setting, we used the whole input XML document as a chunk and only one partial tree was generated from the chunk. Thus, queries are evaluated in serial. The results show that for both datasets, it can process the queries in 100s ms up to several seconds

[1] `http://www.xml-benchmark.org/`

Table 5.12: Statistics of XML dataset.

Datasets	dblp	xmark100	xmark2000	uniprot
Nodes	43,131,420	163,156,531	3,262,490,248	7,891,267,994
Attributes	10,885,411	42,257,706	845,072,591	9,254,412,578
Values	39,642,166	67,254,767	1,344,932,943	1,490,598,653
Total	93,658,997	272,669,004	5,452,495,782	18,636,279,225
# of distinct tags	47	77	77	82
Size (byte)	1,912,866,012	11,758,954,863	236,138,315,428	383,954,056,809
Depth	6	13	13	7

Table 5.13: Queries used in the experiments.

Name	Dataset	Query
XQ1	xmark	/site/closed_auctions/closed_auction[annotation/ description[text/keyword]]
XQ2	xmark	/site//keyword/ancestor::mail
XQ3	xmark	/site/open_auctions/open_auction /bidder[1]/increase
XQ4	xmark	/site/people/person/name/following-sibling::emailaddress
XQ5	xmark	/site/open_auctions/open_auction[bidder /following-sibling::bidder]/reserve
DQ1	dblp	/dblp//i/parent::title
DQ2	dblp	//author/ancestor::article
DQ3	dblp	/dblp//author/following-sibling::author
DQ4	dblp	//author[following—sibling::author]
DQ5	dblp	/dblp/article/title/sub/sup/i/following::author
UQ1	uniprot	/entry[comment/text]/reference[citation /authorList[person]]//person
UQ2	uniprot	/entry//fullName/parent::recommendedName
UQ3	uniprot	/entry//fullName/following::gene
UQ4	uniprot	//begin/ancestor::entry
UQ5	uniprot	//begin/parent::location/parent::feature/parent::entry

Table 5.14: Evaluation by one EC2 instance

Dataset	xmark100					dblp				
Loading (s)	222					47				
Memory (GB)	8.5					3.1				
Query	XQ1	XQ2	XQ3	XQ4	XQ5	DQ1	DQ2	DQ3	DQ4	DQ5
Time (ms)	591	1888	494	1771	1784	11	786	1863	3254	602

Table 5.15: Evaluation by multiple EC2 instance

Dataset	xm2000					unirpot				
Loading (s)	210					379				
Memory (GB)	173					560				
Query	QX1	XQ2	QX3	QX4	QX5	UX1	UX2	UX3	UX4	UX5
Time (ms)	5951	819	1710	1168	3349	2573	2408	1324	5909	6220

for the queries used on xmark100 and dblp.

Evaluate Queries on Multiple EC2 Instances

In this experiment, we investigate the query performance processing 100s GB of XML document on 32 EC2 instances. We use UniProtKB and XMark2000(f = 2000) as experimental data. The results are shown in Table 5.15. Each instance holds only one work and each worker was in charge of a single partial tree.

In the parsing phase, for 0.545 billion and 1.86 billion elements, each of which takes 31 bytes, the memory consumption should 157 GB and 537 GB respectively. The experimental results show the memory consumption are 173 and 560 GB, which are close to our analysis. The overheads is some intermediate data generated during construction. The parsing times as shown in Table 3 are relatively short with regard to the data sizes. The evaluating results for XQ1 to XQ5 in Table 3 show that the query times are just a few seconds for evaluating 220 GB and 358 GB XML data. Besides, the loading times are just 210s and 379s. The throughput is around 1 GB/s. For comparison, PP-Transducer [80] achieved the best throughput of 2.5 GB/s by using 64 cores. Although it is faster than ours, the queries we can process are more expressive than PP-transducer, which does not support order-aware queries.

5.6.2 Scalability

This experiment is used to explore the scalability of our indexing scheme with up to 64 workers on 8 computing instances, each of which has 8 virtual cores and thus runs 8 workers.

Dataset and XPath Queries

In our experiment, we set f to 160 for xmlgen to generate an 18.54 GB of XML document xmark160, which has 267 M element nodes, 61.3 M attribute nodes and 188 M content nodes, totally 516.3 M nodes. We used 7 queries QX11 – QX17 to evaluate the scalability of our indexing scheme. These queries include commonly used axes with predicate as shown in Table 5.16.

Table 5.16: Queries used for xmark160 dataset.

querykey	query	hit nodes
QX11	/site/open_auctions/open_auction/bidder/increase	9577159
QX12	/site//keyword	11271671
QX13	/site//keyword/parent::text	6503643
QX14	/site//text[./keyword]	6503643
QX15	/site/people/person[./profile/gender]/name	1022629
QX16	/site/people/person/name/following-sibling::emailaddress	4080000
QX17	/site/open_auctions/open_auction[./bidder /following-sibling::annotation]/reserve	1734198

Hardware

The hardware we used were Amazon Elastic Compute Cloud (EC2) M5 Instances[1]. M5 Instances are general purpose compute instances that are powered by 2.5 GHz Intel Xeon Scalable processors and offer a balance of compute, memory, and networking resources for a broad range of workloads. We used m5.2xlarge in our experiment, which has 8 virtual cores, equipped with 32 GB of memory and supported with solid state drives. The instance runs Amazon Linux AMI 2018.03.0 (HVM) that supports Java by default. The Java version we used was "1.7.0_181", OpenJDK Runtime Environment

[1] `https://aws.amazon.com/ec2/instance-types/m5/`

(amzn-2.6.14.8.80.amzn1-x86_64 u181-b00) OpenJDK 64-Bit Server VM (build 24.181-b00, mixed mode). The network among EC2 instances was a local network and the network speed is 1 gbps.

Comparison with BaseX

In the experiment, we evaluate queries on a single worker in conparision with BaseX, the state-of-the-art XML database engine. The version of BaseX we used was 8.6.7 implemented on java 1.7. We ran it in server/client mode. A BaseX server and a BaseX client were running on two EC2 instances (one as master and the other as worker). A database was created from xmark160 by the BaseX server. The server was set in main memory mode and turned text indexing off for the creation to make both BaseX and ours in the same setting.

To evaluate queries, we used the following XQuery expression for BaseX: `for $node in db:open('xmark160')&query return db:node-pre($node)`, where `&query` represents an XPath query. This expression returns a list of PRE-values and may take a lot of time for sending and receiving among the BaseX server and a BaseX client. Since network part is not what we are interested, we applied count() function to the results of the XPath query to both BaseX and ours so that the final outcome will be only an integer to be returned over network, removing the time cost of network.

As shown in Figure 5.16, our indexing outperformsBaseX for all the queries. Most of the queries takes only one half or one third time compared with that of BaseX. The most significant one is QX12, for which ours is over 13 timers faster than BaseX. This is because in the two steps of Q2, `/site` returns only 1 node, taking negligible time, while the second step, `//keyword`, can greatly utilize the grouped-array to skip evaluating most irrelevant nodes with different tag names as `keyword`, thus achieving the best performance.

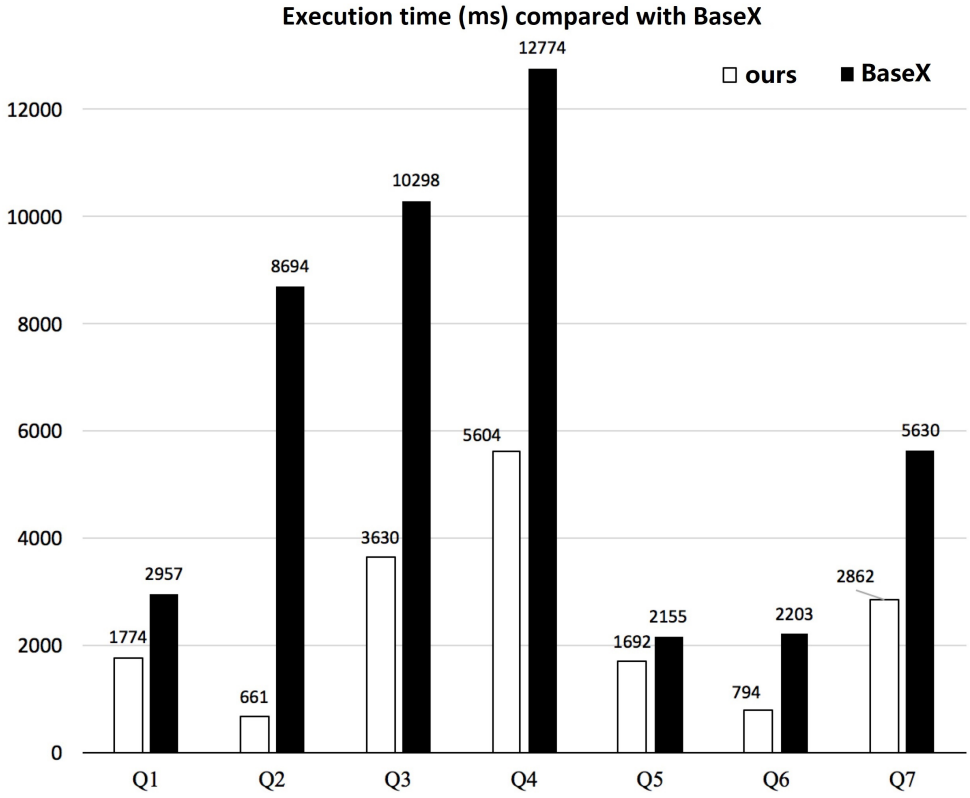


Figure 5.16: Execution time (ms) compared with BaseX.

Processing Queries in Parallel With multiple Workers

This experiment is to test the speedups of our indexing scheme with multiple workers. In this experiment, we use 1 instance as master to control query processing and up to 8 instances for workers to execute queries. Since the instance m5.2xlarge has 8 virtual cores, we arranged at most 8 workers on a single instance. Given 8 instances, there are totally 64 workers involved in the computation, as well as 64 chunks we divided at most. Due to the imbalance of xmark160, not all the worker may have hit nodes of running queries. Thus, for simplicity of discussing, we call workers who have hit nodes *active workers*, while for the rest *idle worker*.

The XML dataset xmark160 is divided into different number of chunks, to be processed by different numbers of workers on up to 8 instances. From each chunk, a partial tree will be created, which will be possessed and processed by a single worker (we assign

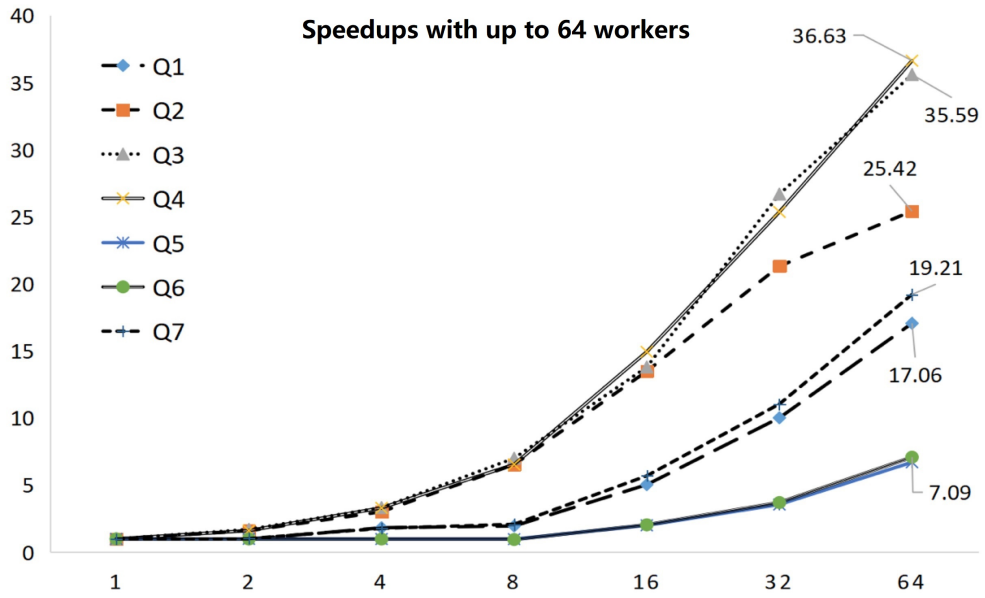


Figure 5.17: Speedups with up to 64 workers.

only one chunk to a worker in this experiment).

To achieve better load balance, we used cyclic distribution to assign chunks to instances. This means that we assign chunks to each instance, making consecutive chunks be assigned to different instances. For example, given 8 chunks, $chunk_1$, $chunk_2$, ..., $chunk_9$, and 4 computing nodes, com_1, com_2, ..., com_4, we assign them as $com_1(chunk_1, chunk_5)$, $com_2(chunk_2, chunk_6)$, $com_3(chunk_3, chunk_7)$, $com_4(chunk_4, chunk_8)$. In such order, we can make the workers utilize the resources of computing nodes.

We record the wall-clock time form the master's side. The timing starts from the master sending a message to all workers to start a query, and ends at the moment when the master receives the work-done message from the last worker, denoting querying work is complete. The execution times are listed in Table 5.17.

Observations

From the results, we have the following observations.

Execution Time Reduced with More Workers

From the results in the table, it is clear that with the number of workers increased,

Table 5.17: Execution time in milliseconds

# of workers	1	2	4	8	16	32	64
QX11	1774	1789	968	905	352	177	104
QX12	661	410	219	101	49	31	26
QX13	3630	2120	1090	518	263	136	102
QX14	5604	3467	1695	855	375	221	153
QX15	1692	1685	1709	1731	841	475	252
QX16	794	800	803	834	388	214	112
QX17	2862	2817	1595	1369	502	259	149

the execution times of most queries are reduced. For example, the execution time is nearly halves every time when the number of workers doubled for Q3. It clearly showed that the parallel processing of XPath queries using our index is efficient to reduce execution time by using more workers.

Imbalance of XML Document Can Prevent Speedups

As we can also notice, however, there are some cases execution times do not reduce at all even with more workers. For example, no matter the number of workers is 1, 2, 8 or 8, the execution times of all the cases are still basically the same, such as around 1700 ms for QX15. We analyzed the number of active workers and found that this is caused by the imbalance of XMark datasets. In the xmark160, the hit nodes of queries may only reside on a consecutive part of the XML document. Then, after being divided, the hit nodes may be distributed in a small number of chunks. Thus, only the workers that possess these chunks can be active, while the rest workers just stay idle. Let us continue to take QX15 as an example. As the number of workers increased from 1 to 8, the number of active nodes, however, did not increase and still stays 1, i.e. there was only one active worker for the query. Therefore, with only one worker, it takes nearly the same amount of time to process the same amount of hit nodes regardless how many workers were totally used.

Imbalance of XML Document Can Spoil Speedups

We also notice that some execution times are not reduced much when even when more active workers involve. For example, QX12 takes 661 ms by one active worker and 410 ms for two active workers, the execution was reduced but not halved. This

is because the hit nodes did not evenly distributed over the two active workers. As we investigated, one worker took 406 ms to collect 6785094 hit nodes, while another worker took 256 ms for 4486577 hit nodes. Therefore, the speedup is degraded due to the imbalanced distribution of hit nodes over chunks. Thus, we can learn that the imbalance can not only prevent the speedup, but also can degrade the speedup.

To understand how much speedup ours can achieve, we use the execution time done by a single worker as baseline. The results are shown in Figure 5.17. From the figure, it shows that ours can achieve better speedup when using more workers. The best speedup is achieved by QX13 with a factor of 36.63. We also notice that when the number of workers are large than 8, the speedup of most queries becomes dramatic, which means with more chunks divided, the imbalance can be smoothed and it can be more effective for better speedups. This is because with smaller chunks, hit nodes can be well distributed to more chunks; meanwhile, with the cyclic distribution of these small chunks, we can have more active workers to participate in the querying process, thus achieving better speedups.

Further Discussion

From the experiment results and our analysis in previous sub section, we can learn that it is better to assign more chunks to a worker rather than one as what we conducted in the experiment. In such case, more workers will be involved to possess chunks that contain hit nodes. In ideal case, hit nodes consecutively contained in a part of the XML document can be divided into more chunks, and these chunks can be distributed to each of workers. Then, all the workers become active and we can achieve better speedups. However, with too many chunks, it is not clear how much overhead on memory, thread or network etc will be involved. We need to find a trade-off about the maximum number of chunks to reach the best performance. Thus, it is worth studying for the future work.

5.7 Summary

In this chapter, we have first developed a novel tree structure called partial tree based on a vertical fragmentation of XML documents. To construct partial trees from

an XML document, there are four phases. First, we split the XML document into several chunks. Second, we parse the chunks in parallel to form several sets of subtrees that have some open nodes. Third, we compute pre-paths from all open nodes. Last, we put pre-paths to each corresponding set of subtrees to complete the partial tree construction.

After the introduction to partial tree, we have also designed query algorithms for most useful class of XPath queries with three examples to demonstrate how the query algorithms work.

Last, we have implemented partial tree with a BFS-array based index for high-efficiency. The experiment shows that the implementation reaches a good absolute query time that can evaluate XPath queries over 100s GB of XML documents in 100s millisecond to several seconds.

Chapter 6

Conclusion and Future Work

This thesis has investigated the parallelization of XPath queries on large XML documents in two ways, a data partition strategy on top of modern XPath processors and a novel data structure partial tree. We conclude our thesis in this chapter.

6.1 Conclusion

Parallelization of XPath queries on XML documents has been studied in the past decade. Most of these studies either focused a small set of XPath queries or were not practical for large XML documents. Thus these studies cannot meet the requirements of the rapid grow of XML documents.

To overcome the difficulties, we first revived an existing study proposed by Bordawerker et al. in 2008. Their work was implemented on Xalan, which is a XSLT processor and has already been out of date now because the hardware and software have both changed. We presented our three implementations on top of a state-of-the-art XML database engine BaseX over XML documents sized server gigabytes. Since BaseX provides full support for XQuery/XQuery 3.1, we can harness this feature to process sub-queries from the division of target XPath queries.

Through our implementations, we are the first to experimentally prove that it is possible to obtain significant speedups by simply rewriting queries into sub-queries and

parallelizing the evaluation of them on top of an XML database engine over gigabytes of XML documents, without need to modify source code of the engine. From the experimental evaluation, our implementations exhibited a great advantage that we are able to use off-the-shelf XML database engines to parallelize the evaluation of XPath queries over gigabytes XML documents, which is very convinient and practical. For processing larger XML documents, we presented a proposal to extend the study to distributed-memory environment by introducing fragmentation that divides an input XML document into multiple fragment containing information for later querying.

For processing larger XML documents with more expressive expressions, we proposed a novel tree, called partial tree. With partial tree, we extend the processing of XML documents from shared-memory environments to distributed-memory environments, making it possible to utilize computer clusters. We also proposed an efficient BFS-array based implementation of partial tree. The experiment results showed the efficiency of our framework that the implementation are able to process 100s GB of XML documents with 32 EC2 computers. The execution times were only seconds for most queries used in the experiments and the throughput was approximately 1 GB/s. There is only one known study that reached faster throughput than ours, which was 2.5 GB/s with 64 cores. However, ours can support more complicated queries, such as order-aware queries, thus making our approach more expressive.

Besides the throughput, partial tree also has the following two good features. First, it is practical to evenly divide an large XML document and create partial trees out of the similar sized chunks so that we can reach good load-balance. Second, partial trees are portable in both shared-memory environments and distributed-memory environments. This means that we can make them work in both environments without changing the setting of partial tree. Therefore, partial tree is a promising data structure and helpful for parallel XML processing, especially for large XML documents in distributed-memory environments.

6.2 Future Work

Based on the studies of BaseX and the partial tree, there are several works that are worth doing in the future.

Firstly , although partial trees are suitable for processing XPath queries over large XML documents, the functionality and fault tolerance, which are both important for process large XML documents, are still weak when partial trees work alone. Therefore, developing a partial tree based framework that cooperates with the distributed systems such as MapReduce or similar frameworks would be a good designing choice. Also, equipping additional programming interfaces to handle more complicated queries or larger data is practically important for the framework.

Secondly, the application of partial tree to BaseX would also be an interesting work. By the introduction of partial tree into BaseX, we can exploit good features of partial tree, such as handling imbalanced trees. In this way, it is high likely to achieve good scalability, especially in case of the implementation of BaseX in distributed-memory environments.

Third, although we have presenteda proposal about the parallelizaton of XPath queriesby horizontal fragmentation on BaseX, we have not experimentally studied the performance of it. It is thus worth conducting experiment for this purpose.

Lastly, the optimization of both studies for the performance is worth intensively studying. There are some factors that limit their performance, such as network, I/O, software/hardware settings, etc. By investing them, new approaches or better algorithms should be designed and proposed.

6.3 A Proposal To Extend Our Appraoch To Distributed-Memory Environments

The following section is an ongoing work that will be done in the future. In Chapter 4, we have introduced our implementations on top of BaseX and the evaluation with a single BaseX server on a dual-CPU system. As introduced in the original study, data

partitioning strategy was 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 [21], the authors had pointed out that the parallelization model is over XML data model, and it can also be adapted to any XML storage layout. However, no matter in the original study, or in our previous study, the strategies are applied both in a shared-memory environment.

Based on our previous experiment results, it would also be promising to use multiple BaseX servers on multiple CPUs in distributed-memory environments over large XML documents so that we can exploiting computer clusters to process them more efficiently. Then, here comes a question: how we can apply it in a distributed-memory environment over a number of computers? In this study, by exploiting horizontal fragmentation on XML data, we present a proposal to apply data partitioning to distributed-memory environments with XML fragmentation.

6.4 Fragmentation

6.4.1 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 [58, 59]. The most common XML fragmentations are horizontal fragmentation and vertical fragmentation [59]. Due to the independence nature of horizontal fragments, it is relatively a more direct and practical way to work together with data partitioning. We thus focus on only horizontal fragmentation.

6.4.2 Definitions

To process a large XML document in a distributed-memory environment, we first need to divide an XML document into multiple fragments to be allocated to multiple computing node for querying. To make the fragments well balanced, we introduce a fragmentation algorithm with an example in this section.

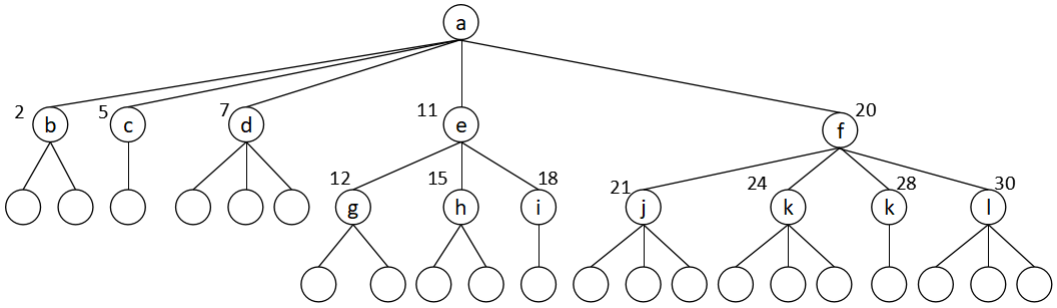


Figure 6.1: An example tree and the PRE values along with nodes

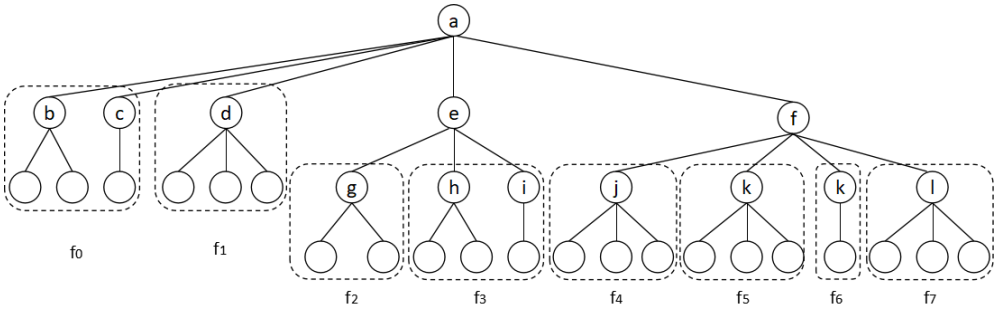


Figure 6.2: Fragmentation on the example tree given maxsize = 5.

Fragment

To begin with, we give definition to fragment first. In this study, a *fragment* is a collection of subtrees. Since the main purpose 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. Therefore, a fragment satisfies two conditions. Firstly, the roots of subtrees in a fragment are consecutive children of a single node in the input tree. Secondly, the number of elements in a fragment is less than or equal to a given integer *maxsize*. Let us take the tree in Fig. 6.1 as an example. Let maxsize be 5, then we have the fragments shown in Figure 6.2. After applying horizontal fragmentation to the example XML tree, we obtain eight fragments enclosed in dotted rectangles.

Anchored Fragment

In order to ease performing top-down XPath queries, we augment each fragment with a path from the root of the whole tree to the subtrees. We call this augmented

fragment *anchored fragment*. In the running example, we have eight anchored fragments as shown in Figure 6.3.

Root-merged Tree

In most existing XML database management systems, such as BaseX, it takes a single XML tree to create an XML database instance. Since a large number of databases instance arises overheads, we reduce the number of database instances by merging some of anchored fragments to *root-merged tree*. A root-merged tree is a number of anchored fragments merged at the root. we merge the root node of anchored fragments only, which is enough to make a list of fragments into a tree.

To make the merged trees work more size-balanced, we apply the following two rules to anchored fragments when merging. Firstly, anchored fragments are randomly grouped. Each group contains a similar number of anchored fragments and. Then, from each group, a single root-merged tree is created by mering the root node of the anchored fragment in the group. Secondly, the anchored fragments of a root-merged tree are ordered in the original order e.g. document oder, on purpose of simplifying the process of reordering results.

For the running example, we construct four root-merged trees from eight fragments with randomization and re-ordering as shown in Figure 6.4.

Fragment Index

In order to facilitate query evaluation, we design *fragment index* that is the information required or useful for managing fragments. It has the following items for each fragment.

- fid: fragment ID.
- mid: the root-merged tree that has the fragment
- mrank: the rank (position) of the fragment in the root-merged tree
- size: the size of fragments

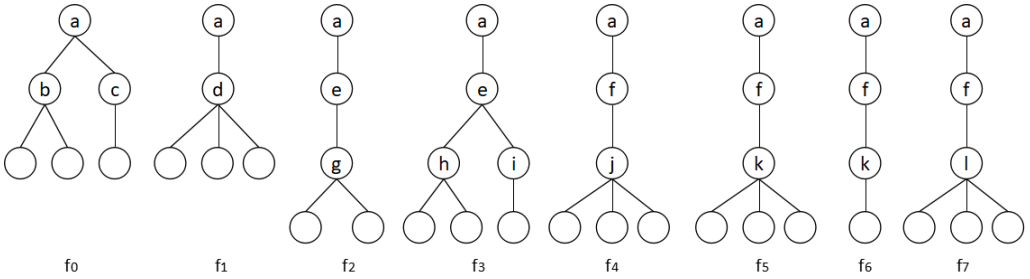


Figure 6.3: Anchor trees.

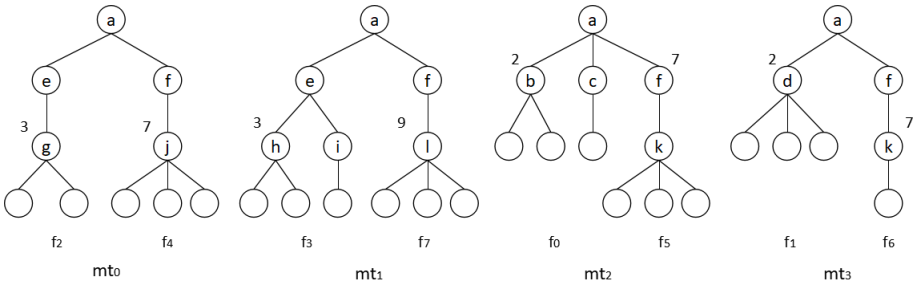


Figure 6.4: Root-merged trees.

- gpre: pre-index (in the input tree) of the first element
- mpre: pre-index (in the root-merged tree) of the first element

For the running example, the fragment index of is created as shown in Table 6.1.

Pruned Tree

In our design, we can perform any XPath queries inside of a fragment, but we need careful computation for XPath queries that may go out of the fragment. In order to perform those XPath queries, we need to know the global tree structure in which fragments are located. For this purpose, we construct a *pruned tree* by replacing the subtrees with a single node for each fragment. In our running example, we have a pruned tree as shown in Figure 6.5.

In order to compute the PRE index of the input tree easily, we add *gpre* for each node. For the nodes representing pruned parts, we add *fid* to link to the fragmentation index.

fid	mid	mrank	size	gpre	mpre
0	mt2	0	5	2	2
1	mt3	0	4	7	2
2	mt0	0	3	12	3
3	mt1	0	5	15	3
4	mt0	1	4	18	7
5	mt2	1	4	21	7
6	mt3	1	2	24	7
7	mt1	1	4	30	9

Table 6.1: Fragment index.

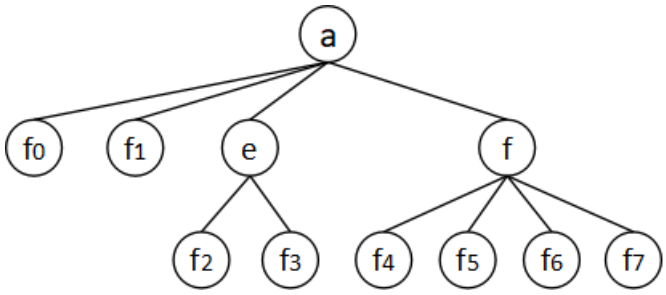


Figure 6.5: Pruned tree.

6.4.3 Fragmentation Algorithm

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 describes how our fragmentation works to apply a horizontal fragmentation to an input 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. In Line 1–2, we declare an empty list of fragments and a new fragment for holding results. In Line 3–17, we traverse nodes in the input list to generate fragments. If a node have more number of descendant nodes greater than maxsize, we apply the fragmentation literately on the children of the node (Line 4–9) and add the results into $fragments$. If not, we check the total number of descendant nodes in

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

```
1: fragments ← [] //a list of fragments
2: fragment ← NEWFRAGMENT() // create a new fragment
3: for all node ∈ nodes do
4:    if size(node) > maxsize then
5:       if fragment.subtrees.size() > 0 then
6:          fragments.Add((fragment))
7:          fragment ← newNEWFRAGMENT()
8:       end if
9:       fragments.AddAll(FRAGMENTATION(child(node), maxsize))
10:   else if NodeSize(node) + NodesSize(fragment.subtrees) > maxsize then
11:      fragments.Add(fragment)
12:      fragment ← NEWFRAGMENT()
13:      fragment.subtrees.Add(node)
14:   else
15:      fragment.subtrees.Add(node)
16:   end if
17: end for
18: for i ∈ [0, fragments.length) do
19:    fragments[i] ←AddPath(fragments[i])
20: end for
21: return fragments
```

Figure 6.6: The fragmentation algorithm.

Algorithm 2 MakeAnchoredFragment(*fragment*)

Input: *fragment*: a fragment
Output: an anchored fragment augmented with the path to the root of the whole tree

```
 1: p ←parent(fragment.subtrees[0])
 2: node ← clone(p)
 3: node.addChildren(subtrees)
 4: while parent(p) ≠ NULL do
 5:    p ← parent(p)
 6:    tempnode ← clone(p)
 7:    tempnode.addChild(node)
 8:    node ← tempnode
 9: end while
10: fragment.root ← node
11: return fragment
```

Figure 6.7: Add path to a list of subtrees to create a anchored fragment

$fragment.subtrees$ and the *node* excesses maxsize, we save the current fragment and put the current node into a new fragment (Line 10–13). Otherwise, the current node is added to $fragment.subtrees$ as one of the subtrees in the current fragment. After the iteration, we obtain a list of fragments, each of which contains a list of subtrees. We add the last use Algorithms 2 to create anchored fragments by adding the path from the current subtrees to the root of the original tree (Line 18–20). 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. We describe them as below:

- `NodeSize(`*node*`)` returns the number of descendants of *node*, where node is a single node.
- `NodesSize(`*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*.

6.5 Our Distributed 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 master and N_s workers. The master is a computer running a Java program that is the implementation of our query algorithm. It works for sending queries to all workers and processing results returned from them. A worker is a computer that runs a BaseX server in charge of evaluating received queries. After fragmentation, the fragments are mapped to workers. For each workers, a root-merged tree is assigned and accordingly a database instance will be created on every worker. Then, queried are evaluated on them and the results returned from all the workers will be merged on the master. To compare with, we introduce both cases that the data partitioning strategiesis used or not. The one with our data partitioning is the regular query and the other is with data partitioning.

6.5.1 Query Rewriting

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

Regular Query

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]
```

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

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

Query with Data Partitioning

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 6.5.1.

6.5.2 Evaluating Queries

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

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

6.5.3 Processing Results

The results are returned from all servers through the network (some workers 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 master (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 workers 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 partitioningstrategy is used.

Regular Query

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 GETMPRE($f$) that returns the *mpre* of the fragment f and the PRE value *pre* of a node n, if GETMPRE(f_i) $\leq pre <$ GETMPRE(f_{i+1}), ($i <$ n - 1) or 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.

Query with Data Partitioninng

Since data partitioningstrategy 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.

Bibliography

[1] Apache spark. http://spark.apache.org/.

[2] Extensible markup language (XML) 1.0 (fifth edition). https://www.w3.org/TR/REC-xml/.

[3] Simple api for xml. https://en.wikipedia.org/wiki/Simple_API_for_XML.

[4] Size of wikipedia. https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia.

[5] Wikipedia. https://en.wikipedia.org/wiki/Main_Page.

[6] Wikipedia dump service. https://dumps.wikimedia.org/.

[7] Xalanprocessor. https://xml.apache.org/xalan-j/.

[8] Xml schema. https://www.w3.org/standards/xml/schema.

[9] Xquery 3.1: An xml query language. https://www.w3.org/TR/xquery-31/.

[10] Basex officail documentation. http://docs.basex.org/, 5 2017.

[11] Foto N. Afrati, Matthew Damigos, and Manolis Gergatsoulis. Lower bounds on the communication of XPath queries in MapReduce. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference*, pages 38–41. CEUR-WS.org, 2015.

[12] Shurug Al-Khalifa, H.V. Jagadish, Nick Koudas, Jagnesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient xml query pattern matching. In *Proceedings of the 12th International Conference on Data Engineering*, pages 141–152. ACM, 2002.

[13] Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. Qrs: a robust numbering scheme for xml documents. In *the 19th International Conference on Data Engineering (ICDE03)*, pages 705–707, 2003.

[14] Alexandre Andrade, Gabriela Ruberg, Fernanda Baio, Vanessa P. Braganholo, and Marta Mattoso. Efficiently processing xml queries over fragmented repositories with partix. In *Current Trends in Database Technology*, pages 150–163, 2006.

[15] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, and Demian Raven. Xmltk: An XML toolkit for scalable XML stream processing. Technical report, Department of Computer and Information Science , University of Pennsylvania Scholarly Commons, 2002.

[16] Nicole Bidoit, Dario Colazzo, Noor Malla, Federico Ulliana, Maurizio Nolè, and Carlo Sartiani. Processing XML queries and updates on Map/Reduce clusters. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*, pages 745–748. ACM, 2013.

[17] Leykun Birhanu, Solomon Atnafu, and Fekade Getahun. Native XML document fragmentation model. In *Sixth International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2010, Kuala Lumpur, Malaysia, December 15-18, 2010*, pages 233–240, 2010.

[18] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*, pages 479–490. ACM, 2006.

[19] Angela Bonifati and Alfredo Cuzzocrea. Efficient fragmentation of large XML documents. In *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, pages 539–550, 2007.

[20] Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, and Bryant Wei-Lun Kok. Statistics-based parallelization of XPath queries in shared memory systems. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*, pages 159–170. ACM, 2010.

[21] Rajesh Bordawekar, Lipyeow Lim, and Oded Shmueli. Parallelization of XPath queries using multi-core processors: Challenges and experiences. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, pages 180–191. ACM, 2009.

[22] Dhruba Borthakur. Hdfs architecture guide[j]. `http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`, 2018.

[23] Sujoe Bose and Leonidas Fegaras. Xfrag: A query processing framework for fragmented XML data. In *Proceedings of the Eight International Workshop on the Web & Databases (WebDB 2005), Baltimore, Maryland, USA, Collocated mith ACM SIGMOD/PODS 2005, June 16-17, 2005*, pages 97–102, 2005.

[24] Sujoe Bose, Leonidas Fegaras, David Levine, and Vamsi Chaluvadi. A query algebra for fragmented xml stream data. In *International Workshop on Database Programming Languages*, pages 195–215. Springer, 2003.

[25] Vanessa Braganholo and Marta Mattoso. A survey on XML fragmentation. *ACM SIGMOD Record*, 43(3):24–35, 2014.

[26] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, 2002.

[27] Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu, and Juan A.M. Naranjo. PAXQuery: Parallel analytical XML processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pages 1117–1122. ACM, 2015.

[28] E. Preston Carman Jr., Till Westmann, Vinayak R. Borkar, Michael J. Carey, and Vassilis J. Tsotras. A scalable parallel XQuery processor. In *2015 IEEE International Conference on Big Data (Big Data '15)*, pages 164–173, 2015.

[29] Rongxin Chen, Husheng Liao, and Zongyue Wang. Parallel xpath evaluation based on node relation matrix. *Journal of Computational Information Systems*, 9(19):7583–7592, 2013.

[30] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig^{2}Stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32Nd International Conference on Very Large Data Bases (VLDB '06)*, pages 283–294. VLDB Endowment, 2006.

[31] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed xml documents. In *the 28th international conference on Very Large Data Bases (VLDB' 08)*, pages 263–274, 2008.

[32] Hyebong Choi, Kyong-Ha Lee, Soo-Hyong Kim, Yoon-Joon Lee, and Bongki Moon. HadoopXML: A suite for parallel processing of massive XML data with multiple twig pattern queries. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12)*, pages 2737–2739. ACM, 2012.

[33] Hyebong Choi, Kyong-Ha Lee, and Yoon-Joon Lee. Parallel labeling of massive XML data with MapReduce. *Journal of Supercomputing*, 67(2):408–437, 2014.

[34] Tae-Sun Chung and Hyoung-Joo Kim. Techniques for the evaluation of XML queries: a survey. *Data Knowl. Eng.*, 46(2):225–246, 2003.

[35] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. 2007.

[36] Gao Cong, Wenfei Fan, Anastasios Kementsietsidis, Jianzhong Li, and Xianmin Liu. Partial evaluation for distributed XPath query processing and beyond. *ACM Transactions on Database Systems*, 37(4):32:1–32:43, 2012.

[37] Matthew Damigos, Manolis Gergatsoulis, and Eleftherios Kalogeros. Distributed evaluation of XPath queries over large integrated XML data. In *Proceedings of the 18th Panhellenic Conference on Informatics (PCI '14)*, pages 61:1–61:6. ACM, 2014.

[38] Matthew Damigos, Manolis Gergatsoulis, and Stathis Plitsos. Distributed processing of XPath queries using MapReduce. In *New Trends in Databases and Information Systems: 17th East European Conference on Advances in Databases and Information Systems*, pages 69–77. Springer International Publishing, 2014.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI2004), December 6–8, 2004, San Francisco, California, USA*, pages 137–150, 2004.

[40] Gatan Hains Emil-mircea Andriescu, Anis Azzabi. Parallel processing of forward xpath queries: an experiment with bsml. In *Technical report TR-LACL-2010-11, Universit Paris-Est*, 2010.

[41] Kento Emoto and Hiroto Imachi. Parallel tree reduction on MapReduce. In *Proceedings of the International Conference on Computational Science (ICCS 2012)*, volume 9 of *Procedia Computer Science*, pages 1827–1836. Elsevier, 2012.

[42] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. Indexing highly dynamic hierarchical data. *Proceedings of the VLDB Endowment*, 8(10):986–997, 2015.

[43] Joseph Fong, Francis Pang, and Chris Bloor. Converting relational database into xml document. In *Database and Expert Systems Applications, 2001. Proceedings. 12th International Workshop on*, pages 61–65. IEEE, 2001.

[44] Marcus Fontoura, Vanja Josifovski, Eugene Shekita, and Beverly Yang. Optimizing cursor movement in holistic twig joins. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 784–791. ACM, 2005.

[45] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[46] Torsten Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002.

[47] Sudipto Guha, Nick Koudas, Divesh Srivastava, and Ting Yu. Index-based approximate xml joins. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 708–710. IEEE, 2003.

[48] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, and Rao. Mixed mode xml query processing. *Proceedings VLDB Conference*, pages 225–236, 2003.

[49] Wei Hao and Kiminori Matsuzaki. A partial-tree-based approach for XPath query on large XML trees. *Journal of Information Processing*, 24(2):425–438, 2016.

[50] Hosagrahar V Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks VS Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, et al. Timber: A native xml database. *The VLDB JournalThe International Journal on Very Large Data Bases*, 11(4):274–291, 2002.

[51] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *The 19th International Conference on Data Engineering (ICDE03)*, pages 253–264, 2003.

[52] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. Xparent: An efficient rdbms-based xml database system. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 335–336. IEEE, 2002.

[53] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed xml documents. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 273–284. VLDB Endowment, 2003.

[54] Lin Jiang and Zhijia Zhao. Grammar-aware parallelization for scalable XPath querying. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*, pages 371–383. ACM, 2017.

[55] Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto. Efficient parallel tree reductions on distributed memory environments. In *ICCS 2007: 7th International Conference, Beijing, China, May 27–30, 2007, Proceedings, Part II*, pages 601–608. Springer, 2007.

[56] Takayuki Kawamura and Kiminori Matsuzaki. Dividing huge xml trees using the m-bridge technique over one-to-one corresponding binary trees. *Information and Media Technologies*, 9(3):251–261, 2014.

[57] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An xml indexing structure with relative region coordinate. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 313–320. IEEE, 2001.

[58] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Generating efficient execution plans for vertically partitioned XML databases. *Proceedings of the VLDB Endowment*, 4(1):1–11, 2010.

[59] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee. Scaling XML query processing: Distribution, localization and pruning. *Distributed and Parallel Databases*, 29(5–6):445–490, 2011.

[60] Martin Krulis and Jakub Yaghob. Efficient implementation of xpath processor on multi-core cpus. In *the Dateso 2010 Annual International Workshop on DAtabases (DATESO' 10)*, pages 60–71, 2010.

[61] Sangwook Lee, Jin Kim, and Hyunchul Kang. Memory-efficient query processing

over xml fragment stream with fragment labeling. *Computing and Informatics*, 29(5):757–782, 2012.

[62] Michael Ley. DBLP: Some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500, 2009.

[63] Changqing Li and Tok Wang Ling. Qed: a novel quaternary encoding to completely avoid re-labeling in xml updates. In *Proceedings of the 14th ACM international conference on Information and knowledge management (CIKM '05)*, pages 501–508, 2005.

[64] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, number 361–370, 2001.

[65] Wenxin Liang and Haruo Yokota. Lax: An efficient approximate xml join based on clustered leaf nodes for xml data integration. In *British National Conference on Databases*, pages 82–97. Springer, 2005.

[66] Wenxin Liang and Haruo Yokota. Slax: An improved leaf-clustering based approximate xml join algorithm for integrating xml data at subtree classes. *Information and Media Technologies*, 1(2):918–928, 2006.

[67] Le Liu, Jianhua Feng, Guoliang Li, Qian Qian, and Jianhui Li. Parallel structural join algorithm on shared-memory multi-core systems. In *Ninth International Conference on Web-Age Information Management (WAIM '08)*, pages 70–77, 2008.

[68] Jiaheng Lu, Ting Chen, and Tok Wang Ling. Tjfast: effective processing of xml twig pattern matching. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1118–1119. ACM, 2005.

[69] Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni. Efficient processing of ordered xml twig pattern. In *International Conference on Database and Expert Systems Applications*, pages 300–309. Springer, 2005.

[70] Wei Lu and Dennis Gannon. ParaXML: A parallel XML processing model on the multicore CPUs. *Techincal Report*, 2008.

[71] Robert W. P. Luk, Hong Va Leong, Tharam S. Dillon, Alvin T. S. Chan, W. Bruce Croft, and James Allan. A survey in indexing and searching XML documents. *JASIST*, 53(6):415–437, 2002.

[72] Hui Ma and Klaus-Dieter Schewe. Heuristic horizontal XML fragmentation. In *The 17th Conference on Advanced Information Systems Engineering (CAiSE '05), Porto, Portugal, 13-17 June, 2005, CAiSE Forum, Short Paper Proceedings*, 2005.

[73] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Executing parallel TwigStack algorithm on a multi-core system. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services (iiWAS '09)*, pages 176–184. ACM, 2009.

[74] Kiminori Matsuzaki. *Parallel Programming with Tree Skeleton*. PhD thesis, University of Tokyo, 2009.

[75] Kiminori Matsuzaki and Reina Miyazaki. Parallel tree accumulations on MapReduce. *International Journal of Parallel Programming*, 44(3):466–485, 2016.

[76] Wolfgang Meier. exist: An open source native xml database. In *Net. Object-Days: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 169–183. Springer, 2002.

[77] Jun-Ki Min, Jihyun Lee, and Chin-Wan Chung. An efficient encoding and labeling for dynamic xml data. In *Advances in Databases: Concepts, Systems and Applications (DASFAA '07)*, number 715–726, 2007.

[78] Mathias Neumller. Compact data structures for querying xml. In *Proceedings of the EDBT/ICDT 2002 Joint Conference*, 2002.

[79] Yoshiaki Nomura, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization of XPath queries with tree skeletons. *Computer Software*, 24(3):51–62, 2007. (in Japanese).

[80] Peter Ogden, David Thomas, and Peter Pietzuch. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.

[81] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-friendly XML node labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 903–908. ACM, 2004.

[82] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML data stored in a relational database. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB '04)*, pages 1146–1157. VLDB Endowment, 2004.

[83] Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. A static load-balancing scheme for parallel XML parsing on multicore CPUs. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 351–362. IEEE Computer Society, 2007.

[84] Yinfei Pan, Ying Zhang, and Kenneth Chiu. Parsing XML using parallel traversal of streaming trees. In *Proceedings of 15th International Conference on High Performance Computing (HiPC 2008)*, pages 142–156. Springer Berlin Heidelberg, 2008.

[85] Eugen Popovici, Gildas Ménier, and Pierre-François Marteau. Sirius: a lightweight xml indexing and approximate search system at inex 2005. In *International Workshop of the Initiative for the Evaluation of XML Retrieval*, pages 321–335. Springer, 2005.

[86] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. Twiglist: Make twig pattern matching fast. In *the 12th International Conference on Database Systems for Advanced Applications*, pages 850–862. Springer International Publishing, 2007.

[87] Fatih E. Sevilgena, Srinivas Alurub, and Natsuhiko Futamurac. Parallel algorithms for tree accumulations. *Journal of Parallel and Distributed Computing*, 65(1):85–93, 2005.

[88] David B. Skillicorn. Structured parallel computation in structured documents. *Journal of Univarsal Computer Science*, 3(1):42–68, 1997.

[89] Budi Surjanto, Norbert Ritter, and Henrik Loeser. XML content management based on object-relational database technology. In *WISE 2000, Proceedings of the First International Conference on Web Information Systems Engineering, Volume I (Main Program), Hong Kong, China, June 19-21, 2000*, pages 70–79, 2000.

[90] The UniProt Consortium. UniProt: a hub for protein information. *Nucleic Acids Research*, 43(D1):D204–D212, 2015.

[91] Haixun Wang and Xiaofeng Meng. On the sequencing of tree structures for xml indexing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 372–383. IEEE, 2005.

[92] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media / Yahoo Press, 2012.

[93] Yu Wu, Qi Zhang, and Zhiqiang Yu. Jianhui li. a hybrid parallel processing for xml parsing and schema validation. In *Balisage: The Markup Conference*, 2008.

[94] XMark — An XML benchmark project. http://www.xml-benchmark.org/generator.html.

[95] XML path lanugage (XPath) 3.1. `https://www.w3.org/TR/xpath-31/`, 2015.

[96] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD*

International Conference on Management of Data (SIGMOD '15), pages 17–30. ACM, 2015.

[97] Chun Zhang, Jeffrey Naughton, David Dewitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *Proceedings of the 2001 ACM SIGMOD international conference on Management of data - SIGMOD 01*, pages 425–436, 2001.

[98] Ying Zhang, Yinfei Pan, and Kenneth Chiu. A parallel XPath engine based on concurrent NFA execution. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS '10)*, pages 314–321. IEEE Computer Society, 2010.