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 2017

1. Reviewer:

2. Reviewer:

3. Reviewer:

4. Reviewer:

5. Reviewer:

Day of the defense: Febrary 25, 2018.

Signature from head of PhD committee:

Abstract

In recent decades, the volume of information increases dramatically, 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 very early stage of XML processing, studies mainly focused on the sequential ways, 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 sinlge CPU, but also better availability with cheaper prices. Therefore, parallization become popular in information processing.

Parallelization of XPath queries over XML documents became popular started from the recnet decade. At the time, studies focused on a small set of XPath queires and were designed to process XML documents in a shared-memory environment. Therefore, they are not practical for processing large XML documents, making them difficult 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 Their work was implemented on an XSLT processor Xalan and has already been out of date now due to the developments of hardware and software.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 XQuery/XQuery

3.1, we can harness this feature to process subqueries from the division of target XPath queries.

This pre-hand study establishes the availability of Bordawekar et al's work. Then, we propose a fragmentation approach that can divide an XML document into size-balanced subtrees with randomization for achieving better load-balance. Along with the previous data partitioning strategy, we can process large XML documents efficiently in distribute-memory environments.

The previous partition and fragmentation based study enables us to easily process large XML documents in distribute-memory environments. However, it still has its flasw that it is limited to top-down queries. Therefore, to enrich the expressness of 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 feasuable to utilize computer clusters. We also propose an efficient BFS-array based implementation of partial tree.

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, based on them, we 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 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. Then, we propose an efficient implementation of partial tree. Based on indexing techniques, we developed 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.

Dedication

To Wenjun Xie,
my amazing wife,
who accompanied me
through the most difficult time in my life,
and brought me two beautiful and lovely sons.
My parents, Wenlin Hao and Qinglin Zhu,
who supported and encouraged me
throughout my Ph.D. career.

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 Assoc. Prof. Kiminori Matsuzaki. It is my greatest honor to be his first Ph.D. student. Assoc. Prof. 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 Assoc. Prof. Tomoharu Ugawa for his helpful advice on my research. I would like to thank Prof. Jingzhao Li, who assisted me with some job issues back in China.

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

I would like to thank the following friends: Onofre Call Ruiz, who was my

classmate and lab mate. He gave me a great 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 research opportunity.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Background 1

1.2 A Short Tour 3

1.3 Contributions 6

1.4 Outline 7

2 Related Work **9**

2.1 XML Fragmentation 9

2.1.1 Fragmentation of Trees 9

2.1.1.1 Horizontal Fragmentation 9

2.1.1.2 Vertical Fragmentation 10

2.1.2 Fragmentation of XML Text 10

2.1.3 Holes and Fillers 10

2.2 Parallel XML Processing 11

2.2.1 Tree Accumulation and Reduction 11

2.2.2 XML Streaming 11

2.2.3 MapReduce-based XML Processing 11

2.2.4 Parallel Processing of queries 12

2.3 XML Database Techniques 13

2.3.1 Indexing and Labeling Schemes 13

2.3.1.1 Joins Algorithms 13

3 XML and Query Languages **15**

3.1 XML 15

3.2 XPath Language 17

3.2.1 Definition of XPath 17

3.2.2 Evaluting XPath query 18

3.3 XQuery 19

3.4 Summary 19

4 Parallelization Of XPath Queries on Top of BaseX **21**

4.1 BaseX: An XML Database Engine 22

4.2 Implementing Data Partitioning with BaseX 24

4.2.1 Client-side Implementation 24

4.2.2 Server-side Implementation 25

4.3 Implementing Query Partitioning with BaseX 26

4.4 Integration with Query Optimization 29

4.5 Experiments and Evaluations 31

4.5.1 Experimental Setting 31

4.5.2 Evaluation on Implementations of Data Partitioning Strategy . . 32

4.5.2.1 Total Execution Time 32

4.5.2.2 Breakdown of Execution Time 33

4.5.2.3 Scalability Analysis 35

4.5.3 Evaluation on Implementation of Query Partitioning Strategy . . 36

4.5.3.1 Total Execution Times and Speedups 37

4.5.3.2 Breakdown of Execution Time 39

4.6 Observations and Perspectives 39

4.6.1 BaseX Extensions Desirable 39

4.6.2 Further Perspectives 40

4.7 Summary 41

5 Partial Tree **43**

5.1 Definitions . . . 43

5.2 Characteristics of Partial Tree . . . 44

5.2.1 Properties of Open Nodes . . . 45

5.2.2 Standard Structure . . . 45

5.3 Construction of Partial Trees . . . 46

5.3.1 Construction of Subtrees From Parsing XML Chunks . . . 48

5.3.2 Pre-path Computation . . . 49

5.3.3 Creation of Ranges of Open Nodes . . . 51

5.4 Eavalute XPath Queries on Partial Trees . . . 52

5.4.1 Definitions . . . 53

5.4.2 Queries without Predicate . . . 54

5.4.2.1 Downwards Axes . . . 55

5.4.2.2 Upwards Axes . . . 57

5.4.2.3 Intra-sibling Axes . . . 58

5.4.3 Queries with Predicate . . . 60

5.4.3.1 Preparing Predicate . . . 62

5.4.3.2 Evaluation of Steps within A Predicate . . . 62

5.4.3.3 Processing Predicate . . . 63

5.4.4 Worst-Case Complexity . . . 64

5.5 BFS-array based implementation . . . 66

5.6 Evaluation . . . 71

5.6.1 Datasets and XPath Queries . . . 71

5.6.2 Experimental set-up . . . 72

5.6.3 Evaluate Queries on a Single EC2 Instance . . . 72

5.6.4 Evaluate Queries on Multiple EC2 Instances . . . 72

5.7 Summary . . . 74

6 Conclusion and Future Work **77**

6.1 Conclusion . . . 77

6.2 Future Work 78

Bibliography **81**

List of Figures

1.1 An example XML tree 5
1.2 An example partial tree. 5

3.1 An example of node relationships. 17
3.2 Grammars of XPath queries used for partial tree 18

4.1 List of XPath queries and their partitioning, where pre and suf mean prefix query and suffix query respectively 27
4.2 List of XPath queries used for query partitioning, where [pos] denotes position-based partitioning and {name} denotes branch-based partitioning. 30
4.3 Load balance 36
4.4 Increase of work 36

5.1 Four types of element nodes 44
5.2 The standard structure of partial tree. 45
5.3 an XML tree from the given XML string 47
5.4 Partial trees from the given XML string. 47
5.5 Subtrees from parsing chunks 49
5.6 Overall algorithm of XPath query for partial trees 55
5.7 Query algorithm for downwards axes 56
5.8 Query algorithms for upwards axes 60
5.9 Algorithm for Following-sibling axis 61
5.10 Query algorithm for handling predicate 65

LIST OF FIGURES

5.11 Query algorithm for child axis in a predicate 66
5.12 An example XML tree with values. 68
5.13 Partial trees with values from the XML document. 69
5.14 A partial tree and its representation with two arrays 70

List of Tables

4.1 Summary of execution time of data partitioning 32
4.2 Breakdown of execution time for client-side implementation 33
4.3 Breakdown of execution time for server-side implementation 34
4.4 Summary of total execution times(ms) of queries by query partitioning . 37
4.5 Breakdown of execution time (ms) 37

5.1 Open node lists 51
5.2 Results of pre-path computation in AUX 51
5.3 All open nodes 52
5.4 Open node lists with ranges 52
5.5 Evaluating downward steps of Q1 57
5.6 Evaluating the last step of Q1 59
5.7 Evaluating the location steps of Q2 61
5.8 Prepare the predicate of Q3 62
5.9 Evaluate inner steps of the predicate in Q3 64
5.10 Process the predicate in Q3 64
5.11 Time Complexity 67
5.12 Statistics of XML dataset. 73
5.13 Queries used in the experiments. 73
5.14 Evaluation by one EC2 instance 74
5.15 Evaluation by multiple EC2 instance 74

Chapter 1

Introduction

1.1 Background

In the information enginering, XML processing as a common and popularly used information processing technology, studies how to process XML documents and has been intensively studied. In recent decades, the volume of information increases dramatically, leading an urgent demand for high-performance data processing technologies. This change also had a greate influence to XML processing technology that sizes of XML documents are also increasing dramatically. For example, Wikipedia [**?**] provides a dump service [**?**] that exports wikitext source and metadata embedded in XML documents. The sizes of the data was less than one gigabyte before 2006. In just 10 years, it increased to over 100 gigabytes nowadays [**?**]. Some XML documents even reach hundreds of gigabyte. For example, an online database of protein sequence UniProtKB [**?**] can store all the data in a single XML document sized 358 GB.

Up to the beginning of this century, studies of XML processing mainly focused on sequential processing [**?** **?** **?** **?** **?**], due to that commonly used CPUs were mostly single-core processors. As the multiple-core CPUs gradually became dominant gradually, more researches shifted to parallel XML processing and thus more related studies were proposed [**?** **?** **?** **?**].

One common topic in the field of parallel XML processing is the parallelization of XPath [**?**] queries over XML documents. The basic idea of parallelize is to divide an

XML document into multiple parts and process queries separately in parallel. One key problem of the parallelization is how to deal with the hierarchical structure that is the intrinsic characteristic of each XML document. This characteristic plays a very important role in parallel XML document processing, forcing us to deal with the connections of the divided parts after splitting.

To address the above difficulty, many approaches had been proposed [**?** **?** **?** **?** **?** **?** **?** **?**]. Most of these approaches tend to represent XML document as trees and preserve them in memory, then parallelizing XML processing on these trees, which usually utilize multi-thread techniques processed in a shared memory where all the threads are able to access the shared XML data. However, these studies discuss the parallelization of XPath queries only on the a whole tree (or a number of whole trees) without considering how to divide them. Thus, they are not suitable for distribued XML document processing.

Another common approach of XML processing is to exploit database techniques. XML processing in databases has also been widely studied [**?** **?** **?** **?** **?**]. Common database techniques, such as indexing [**?** **?** **?**], join algorithms [**?** **?** **?**], are also valid to be applied to XML processing. Although concurrent transactions are available in modern database engines, making it possible and available to be applied in distribued settings, there is, to the best of our knowledge, no existing work that studies the parallelization of XPath queries based on XML databases in a distribued-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 distribted-memory environemts.

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 queries into multiple subqueries, such as [**?** **?**], we can convert the evaluation of the original query to the evaluation of these subqueries. However, there is a technical difficulty as to figure out how to parallelize

the evaluation of an single query by exploiting existing XML database engines, particularly how to distribute an XML document to multiple XML databases and process them efficiently to achieve good performance gain and scalability.

- General Approach to Parallelize Evaluation of XPath Queries.
  When processing XML documents in distributed-memory environments, it is common to partition an XML document into chunks and distribute the processing of chunk to multiple computers. 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 differenties. There are two corresponding approaches proposed for both shared-memory environments and distributed-memory environments, particularly for the latter. In the following section, we present a "short tour" to illustrate the contributions of this thesis.

1.2 A Short Tour

To address the parallelization of XPath queries in XML databases, we demonstrate our two approaches with several examples.

The first approach involves an implmentation work of [**?**] proposed by Bordawekar et al., which presented approaches to easily exploit esxisting XML processors, such as Xalan [**?**], to parallelize XPath queires with no need to modify the processors. Their approach was to partition queries in an ad hoc manner and to merge the results of partitioned queries. Specifically, they proposed three strategies: query partitioning, data partitioning and hybrid partitioning. 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.

Since hybrid partitioning strategy is a mix of the first two partitioning strategies, we focus on the data partitioning and query partitioning strategies in this thesis. For query partitioning, it depends on the relationships of subqueries in the predicate. If we split and-/or-predicates, we have to intersect/union results. For data partitioning, if we perform position-based query partitioning and data partitioning, we have only to concatenate results in order. In our study, we have developed two implementations of data-partitioning parallelization and a implementation of query-partitioning parallelization on top of BaseX [**?**], which is a state-of-the-art XML database engine.

The second approach is based on a novel tree structure, called partial tree. A partital tree is a tree strcutre 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 use in both shared-memory and distributed-memory environments. To understand what a partial tree is and how it works, we use the following XML document as an example.

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

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: *What structure can a single tag represent in the tree*?

For example, consider the underlined part of the document in Figure 1.1. The corresponding nodes of tags in the underlined part are colored gray in the figure. Note that some tags, such as the first `</C>` or the last `<B>` miss their matching tags. Then, how can we represent these tags when we parse this chunk? Besides, A_1 and B_2 are missing in the chunk. Therefore, 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, i.e. A_1 and B_2?

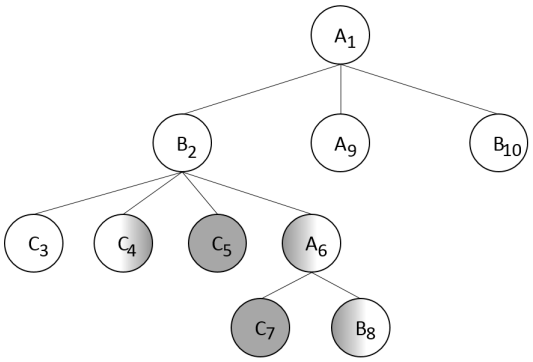


Figure 1.1: An example XML tree

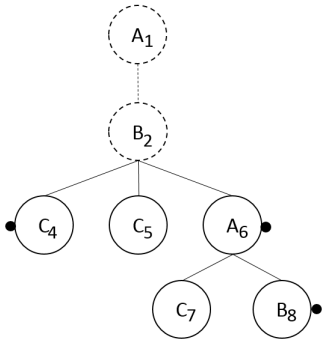


Figure 1.2: An example partial tree.

To address on the above two problems, 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-/distributed- memory environments, it is more specially designed for distributed-memory environments. This is becuase chunks of an XML documents can distribted to multiple computers and then be parsed into partial trees for further parallel processing.

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 [**?**], 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 significant speedups over XML documents sized several gigabytes. Besides the experiment results, we also present significant observations and perspectives from the experiment results.

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 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 indexing techniques, we developed a 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.

1.4 Outline

The thesis is organized in six chapters. An introduction to this study is given in Chapter 1. We review the related work in Chapter 2 and give definitions of XML query languages in Chapter 3. We propose the idea of XPath queries on top of BaseX in Chapter 4. We propose the new tree structure, partial tree in Chapter 5 and we conclude the whole thesis in Chapter 6. Here are the detailed introduction to the following Chapters.

Chatper 2

We dicuss 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 introduced and defined.

Chapter 4

We introduce our implementations of [**?**]. We introduce the XML processing engine BaseX and we present our implementations. We evaluate our implementations on two data sets and propose our observations and perspectives from the experiment results.

Chapter 5

We present a noval tree structure, partial tree. We give the definitions of partial tree, discuss the characteristics and design querying algorithms for partial tree. We also propose an efficient BFS-index based implementation of partial tree. We report the experiment results and analyze the results.

Chapter 6

We summarize this thesis and discuss the future work.

Chapter 2

Related Work

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

2.1 XML Fragmentation

Fragmentation is an important way of efficient processing XML documents, which divides an XML document in fragments and processes them in parallel. Fragmentation is the premise of data-parallel computation algorithms, and has been intensively studied [**?** **?** **?** **?** **?** **?** **?** **?**]. We discuss three ways of fragmentations of XML documents in this section.

2.1.1 Fragmentation of Trees

Data fragmentation is characterized by physical changes to the dataset, that is, the dataset is fragmented and allocated to multiple computational nodes [**?**]. Kling et al. [**?**] modeled fragmentation as horizontal and vertical in terms of XML schema [**?**] that is a language for expressing constraints about XML documents. We now discuss these two fragmentations.

2.1.1.1 Horizontal Fragmentation

Horizontal fragmentation is to divide a document tree into multiple fragments and every fragment follows the same schema of the original XML document. The whole document

is a simple collection of fragments. Horizontal fragmentation is rather straightforward and widely used in parallel XML processing [**?** **?** **?** **?**]. Since the fragments follow the same schema, the queries can be evaluated on them by considering the schema.

2.1.1.2 Vertical Fragmentation

Vertical fragmentation, on the other hand, is a fragmentation that extracts subtrees from the middle of the tree, thus following different schemas. Though this fragmentation does not usually work well for parallel XML processing, it is worth studying. This is because it is a possible and important in case when the data are integrated from different sites or organizations [**?** **?**].

2.1.2 Fragmentation of XML Text

Compared to these tree-based fragmentation techniques, the fragmentation in this study is based on serialized text, which means we cast the partition on the plain text of an XML document in stead of the XML tree parsed from the document. The main advantage of our text-based fragmentation is that we can assign the chunks over distributed file systems [**?**] that are cut by default. Similar idea was introduced by Choi et al. [**?**] in which they added labels to make every chunk a well-formed tree in a preparsing phase.

2.1.3 Holes and Fillers

Bose et al. proposed a fragmentation model [**?**]. Bese on this, Bose et al. proposed a system called Xfrag [**?**]. In this system, an document is divided into multiple sub documents called fillers, where other fragments (the fillers) may fit. Cong et al. [**?**] and Nomura et al. [**?**] adopted a tree-shaped 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. The key part of their approaches to decouple dependencies between evaluations on fragments so as to perform them in parallel.

2.2 Parallel XML Processing

Many existing studies address the topic of XML processing in parallel [**?** **?** **?** **?** **?**]. We discuss parallel XML processing in this section.

2.2.1 Tree Accumulation and Reduction

There are some exsisting ideas of dividing the XML documents and running the computation for trees called tree reduction. Kakehi et al. [**?**] showed a parallel tree reduction algorithm from the nodes in chunks. Based on the idea given by Kakehi et al., Emoto and Imachi [**?**] developed a parallel tree reduction algorithm on Hadoop, and Matsuzaki and Miyazaki [**?**] developed a parallel tree accumulation algorithm. A similar approach was taken by Sevilgen et al. [**?**] who developed a simpler version of tree accumulations over the serialized representation of trees.

2.2.2 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 [**?**] is an XML stream processing tool designed for scalable XML querying. Y-Filter [**?**] applies multiple queries in parallel for a stream of XML data. Among these studies, Ogden et al. [**?**] achieved the highest throughput, 2.5 GB/s, based on the parallel pushdown transducer. Although it is faster than our implementation of partial tree, which is 1 GB/s, the class of queries we support is more expressive than that of PP-transducer, which does not support order-aware queries. In parallel pushdown transducers [**?**], a given document is modeled as a sequence of matched brackets and a fragment is represented as a sequence of unmatched brackets.

2.2.3 MapReduce-based XML Processing

MapReduce [**?**] is a promising approach to large-scale XML processing, which can run on top of clusters of commodity computers. It is suitalbe for scalability as the size of XML data increases very rapidly. Hadoop [**?**], which is a porpular implementtation of

MapReduce, is a common infrastructure for large-scale data processing, and to parallel streaming [? ?]. There have been several studies in this direction [? ? ? ? ? ?]. One of earlier work is by Choi et al. [?] called HadoopXML, which processes XML data in parallel by applying SAX [?] for each chunk. Including this work, most of the existing MapReduce-based frameworks supports a small subset of XPath with `child` and `descendant` axes with predicates [? ? ? ?]. Instead, they extend the expressiveness by the support of some query functionality (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, for example Apache Flink [?].

2.2.4 Parallel Processing of queries

Parallel XML processing has been actively studied after the paper presented by Bordawekar et al. [?], which closely relates to our study. The paper proposes three strategies for XPath queries in parallel: data partition strategy, query partition strategy, and hybrid partition strategy. . In fact, 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 [?]. The main idea in parallelizing XPath queries was to convert XPath queries into (tree) automata [?], 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. [?]. [? ? ?] focus on XPath queries implemented in a shared-memory environment. [?] proposed ideas about XML processing in a forward and forward manner, which is helpful for our research to support backward and upward queries as well. Liu et al. [?] developed a parallel version of structural join algorithm. The study [?] focuses processing a locality-aware partitioning in parallel database systems. Cong et al. [?] formalized parallel processing of XPath queries using the partial evaluation technique: the idea existing behind their partial evaluation is similar to automata.

2.3 XML Database Techniques

2.3.1 Indexing and Labeling Schemes

Indexing is a hot topic for improving the performance of XML queries processing. [**?** **?** **?**] are related to this field. They examined the indices on different types of trees, including B+-tree, R-tree, and XR-tree. O'Neil et al. [**?**] proposed an index called ORDPATH for natively supporting XML data type. 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 very long in case the XML documents are large. Pal et al. [**?**] studied how to improve the query performance by introducing two indexes to nodes and values in SQL Server. Li et al [**?**] improved OrdPath by reducing the length of ORDPATH index when inserting. Min et al. [**?**] proposed an efficient labeling scheme, called EXEL, which incurs no re-labeling of nodes when inserting nodes. Finis et al [**?**] Proposed an idea mainly on how to maintain and query hierarchical data at a high rate of complex, possibly skewed structural updates. There indexes inspire our deisgn of index scheme on XML document to make ours in a more efficient way.

2.3.1.1 Joins Algorithms

Join processing is central to database implementation [**?**]. There are two join algorithms commonly used in XML processing, structural join and twig join.

Structural join [**?**] is mostly based on numbering indexing[**?**], which numbers a nested intervals on nodes and is commonly used in XML and other database applications [**?** **?** **?**]. 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 earily study [**?**] proposed three joint algorithms for processing XML queries, which were similar to structural join. In 2002, Quanzhong Li et al. first proposed structural join in [**?**]. Jiang et al. [**?**] 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. Le Liu et al. [**?**] first applied structural join in parallel over shared-memory environments.

Twig joins are also commonly used for maching a part of an XML documents [**?** **?** **?** **?**]. In twig join, a query is represented as a twig patten, and then is searched on the target XML document. One of the early twig joins was [**?**]. 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 devleoped [**?** **?**]. In 2009, Machdi et al. [**?**] implemented the idea in parallel on multiple cores and in 2012 Choi et al. [**?**] studied the twig joins on Hadoop in a parallel manner.

Chapter 3

XML and Query Languages

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

3.1 XML

XML [**?**] is a standard data describing language used for representing arbitrary data in a hierarchical structure. As introduced, XML documents are increasing dramatically in size over time. Large XML documents in this thesis refer to XML documents whose sizes are greater than 1 gigabytes.

Each XML document corresponds to a logical XML tree (or simply tree) that contains one or more elements. There are several types of elements in the tree of an XML document. In this study, we focus on the following three element types.

- 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. All element nodes are ordered as their corresponding tags appear in the XML document.

- Content node

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

- Attribute node

An attributes node is used to associate name-value pairs inside a start tag of an element node for describing properties of the element node.

Now, we give an example to show what these nodes are. 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 create an XML tree as shown in Figure 3.1 (The example of three element types are shown on left-bottom corner). For example, the node B is an element node. There two nodes below B and are connected to it. The left one is an attribute node with the name 'AT1' and the value 'VAL1'. The right one is a value node with the string value 'TXT1'.

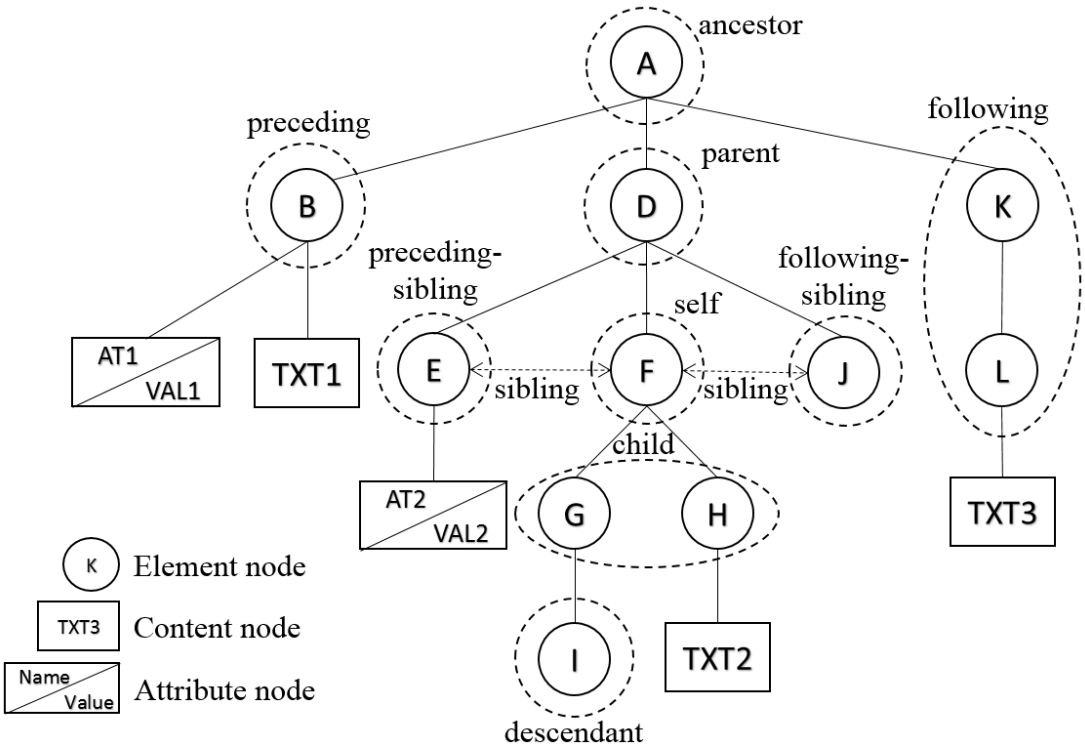


Figure 3.1: An example of node relationships.

3.2 XPath Language

3.2.1 Definition of XPath

XPath is an XML query language used for selecting parts of an XML document [**?**]. XPath queries are represented in path expressions. Each path expression contains one or more *location steps* or simply *stpes*. In this study, each step consists of an *axis*, a *name test*, and at most one *predicate*.

An axis defines the relationships of the current node and the target nodes. 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 selects only attribute nodes, while the other axes select only element nodes. Content nodes can be selected by using function `text()`. Figure 3.1 uses node F as the current node to demonstrate these axes. In the figure, F has a

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 3.2: Grammars of XPath queries used for partial tree

parent D, two siblings: E on the left side as a preceding-sibling and I on the right side as a following sibling, two children: G and H, one descendant I, one preceding B, and two followings: nodes K and L.

A name test is used for selecting nodes. If the name of a tag in an XML document is equal to the name test, the node is selected. 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 "`/descentdant::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 second step.

3.2.2 Evaluting XPath query

When evaluating an XPath query, 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 above to demonstrate. This query first retrieves all the nodes with name `F` in a XML document, and then among their children it retrieves node `H` with one or more children with name `J`. In other words, the result of the query is a set of nodes `H`, each of which has its parent `F` and at least one sibling `G` on its right. The grammar of XPath used in this study for our novel tree structure

partial tree (See Chapter 5) is listed in Figure 3.2.

3.3 XQuery

XQuery is also a language for querying XML documents. It for XML is like SQL for databases. It integrates with XPath language and an XPath query in an XQuery expression returns exactly the same result as in XPath language. XQuery is more expressive and powerful (also more complicated) than XPath. Now, let use consider the following expression as an example. In this study, we used XQuery 3.1 [**?**].

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

This XQuery expression queries an XML document, namely "books.xml". It evaluates an XPath query `/bookstor/book` over the document to select all `book` of the root `bookstore` and returns the `title` of the selected `book`.

3.4 Summary

In this chapter, we have introduced three languages used in this thesis: XML, XPath and XQuery. XML is used to represent arbitrary data. Data stored in XML format are called XML document. XPath and XQuery are languages for querying XML documents. XQuery integreates with XPath and thus is more powerful and expressive.

Chapter 4

Parallelization Of XPath Queries on Top of BaseX

On the topic of parallelelization of XPath queires, one practical and promising approach was proposed by Bordawekar et al. [**?**] in 2009. In this study, three partitioning strategies were presented, i.e. data partitioning, query partitiooning and hybrid partitioing startegies, making it possible to parallelize XPath queries by partitioning an XPath query into subqueries and evaluating them in parallel on XML trees.

However, since this study was based on an XSLT processing, it is thus not clear to the following questions: *(1) Whether and how can we apply their partitioning strategies to XML database engines? (2) What 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 with two large XML documents sized 1.1 GB and 1.85 GB by reviving and extending Bordawekar et al's study. We propose three implementations on the base of the original idea from the paper and we also extended our implementations in term of BaseX by exploiting its optimizations. We experimentally demonstrate the performance on top of BaseX with two large XML documents. The experiment results proved that it is possible to obtain significant speedups by simply rewriting queries into subqueries 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.

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 and an XQuery/XPath 3.1 processor (refer to the official documentation [**?**] for more details). BaesX provides significant features in processing XML data sets. The following are the 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 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 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 the 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> ,
```

where a left superscript denotes a PRE value. Now, consider the following query.

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

The querys selects `book` in 'exampledb' and the result 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 $node
```

The `db:open-pre` takes the database and PRE 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 subarray 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.

BaseX's query optimization is so powerful that 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, as to be described in Section 4.4, is therefore of crucial importance for performance.

Lastly, BaseX can work in a client-server mode. A BaseX server can handle concurrent transactions requested from 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.

[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.

4.2 Implementing Data Partitioning with BaseX

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

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’’, $x)/bidder[last()] ,
```

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 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.

First, we prepare an in-memory database named `‘‘tmp’’` and initialized with `<root> </root>`, which is a temporary database for storing the results of a prefix query. The prefix query is `/site//open_auction` and it selects all `open_auction` and return the PRE values of matched nodes. 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 ,
```

where ⟨⟨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 subarray of `$P`-block partitioned `$arr`. With the array operations of extracting length and subarray, `$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 for data partitioning. It has two-phase, a parallel evaluating phase and a merge phase. In the parallel evaluating phase, a query is divided into multiple subqueries, which are executed by a BaseX server in parallel. In the second phase, the results of all subquery 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 on these groups of nodes and its branches in parallel. Let us take XM5 as example. The children nodes

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::*//title`

of `/site/open_auctions` on 'xmark10.xml' is 120000. Letting P = 3, then we can use the `position` function to divide the query into three subqueries 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 subqueries cover exactly the same parts on the target XML document as that of the original query and returns in turn the same results. Node that since the query is divided by the `position` function in document order, the results are also in document order. Thus, we can simply merge the results orderly to form the final result.

As for the predicate-based partitioning, it is not promising to be introduced in term of XML database engines. This is because it takes extra time to merge the results of subqueries. For example, given a query `/q1[q2 or q3]` that is divided into `q1[q2]` and `q1[q3]`, we can retrieve results from both queries. Since the operation is an '`or`', we need to perform an ordered union to the results of the two subqueries. However, since the resultant nodes in the results, there are two difficulties to merge them. First, we need to determine the order of nodes in the results. Second, the merging process itself is time-consuming when the results are in plain text. Therefore, we do not discuss this partitioning in our study.

We also extend the query partitioning strategy by partitioning the children with different names, called branch-based partitioning. We exploit the structure of the input XML document to evaluate subqueries 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`, 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 the execute time. For example, BaseX optimizes XM3 to

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

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 a step of descendant-or-self axis (`//open_auction`) is replaced with two steps of child axes (`/open_auctions/open_auction`), the search space of this query has been significantly reduced. Note that a more drastic result is observed in XM2 that

, where the attribute index is exploited through function `db:attribute`.

Partitioning strategies convert a given query to two separate ones and therefore affects the capability of BaseX in query optimization. In fact, 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 observed. 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, and it also occurs in query partitioning strategy in term of parallelizing subquereis.

A simple way of resolving this discord is to apply partitioning strategies after BaseX's query optimization. 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 both partitioning strategies to the optimized query, forgetting the original. In fact, XM2–4(c) are instances of such

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`

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.

4.5 Experiments and Evaluations

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

4.5.1 Experimental Setting

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). 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 and decrease the number of packets.

We used two larege XML documents for the experiments: `xmark10.xml` and `dblp.xml`. The dataset `xmark10.xml` is an XMark dataset [**?**] generated with the parameter 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 [**?**], 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

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

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, which are the same as those used in [**?**], and measured execution times 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.

4.5.2.1 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

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

`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 for reference 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 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.

4.5.2.2 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 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

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

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 per se 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 XM queries, 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.

4.5.2.3 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 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

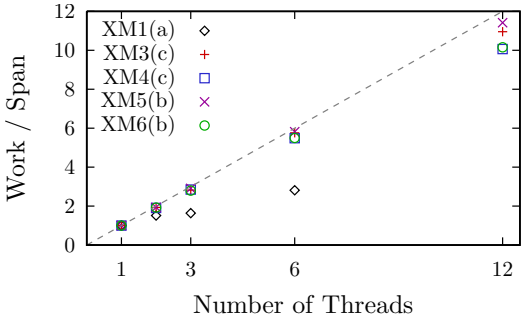


Figure 4.3: Load balance

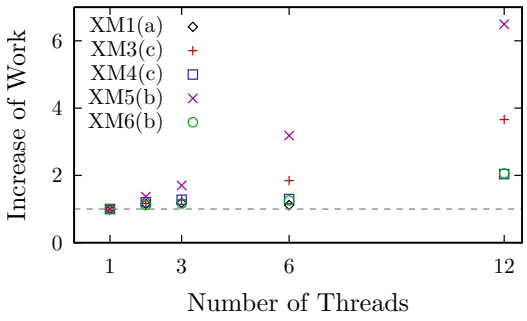


Figure 4.4: Increase of work

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 Figs 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 subquery one by one with a single thread. The "par t_p" columns show the time for executing the x query with one thread and the with 12 or 6 threads depending on

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	Subqueries						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

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.

4.5.3.1 Total Execution Times and Speedups

From Table 4.4, we can see that for most of the queries we have obtained speedups of factors more than 1 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 subqueries 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 subqueries. 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 subqueries of XM2(e) have complete paths. For example, one of its subquery is `/site/regions/africa/item/../parent::item/@id`. Since the full path is contained in the subquires, BaseX can use `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 12000 `open_auction` nodes and evaluates the first 69816 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 need to guarantee the query before the point where partition occurs should be a single path, i.e. the number of nodes that match the query should be one.

4.5.3.2 Breakdown of Execution Time

In this section, we investigate execution times in greater details by analysing the breakdown of execution time. 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 4.5.2.3 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 rather small. 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 paper, we have revived data partitioning [**?**] on top of BaseX and experimentally demonstrated, in the best case of non-trivial queries, 4-fold speedup on a 12-core server. In this section, as concluding remarks, 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 reletively 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) [**?**]. 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 [**?**] will be well-suited but it can incur imbalance in size among fragments. Balanced-size cheap fragmentation based on partial trees [**?**] will be promising for the complement to it. Existing work [**?**] on querying large integrated data will be relevant. Hybrid partitioning [**?**], 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

First, as concluded in [**?**], 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 parititioning, the imbalance of XML documents have a dramatic influence on the query perfomrance 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 greately 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 subqueries can meet the conditions of the BaseX optimizer.

Third, the experiment results clearly show that we can achrive speedup up to 4, which have strongly proved that the partitioning stragegies are available not only on XML trees but also on XML database engines. However, the availability of these stragegies 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 achrive significant performance improvements over the original strategies, such as the server-side implementation of data partitioning strategy.

Chapter 5

Partial Tree

5.1 Definitions

In this section, we introduce our novel tree structure, partial tree. Parital tree is used to process XPath queries over an XML documents in parallel. To use partial tree, we first split an XML document into many chunks and then from these chunks, we can construct multiple partial trees from the chunks. To deeply understand what partial tree is, we first give some definitions for it.

To begin with, we give the defintion of types of element nodes. A partial tree contains four different types of element nodes as shown in Figure 5.1. The four types of element nodes in the figure 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. For a node that does not have both of its tags, we call it *open node*, including left-open node, right-open node and pre-open node. A left-open node has only its start tag, while a right-open node has only its end tag. In case a node loses both of its tags, we call it *pre-node*.

Open nodes are not a new concept. Kakehi et al [**?**] proposed a new approach for parallel reductions on trees and explicitly used 'open' nodes in their paper. Choi et al. [**?**] proposed an idea to label nodes in split chunks of a large XML documnet. Although they did not use the term, but the idea is exact the same as that of open nodes.

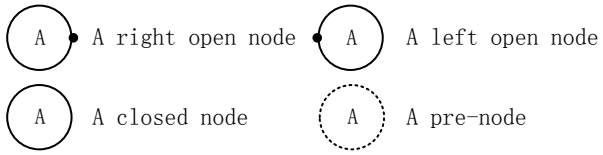


Figure 5.1: Four types of element nodes

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 becuase 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 subtrees parsed from a chunk, which specifies the relationships between the root of the whole tree and the substrees.

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); (2) all open nodes, including pre-node, are element nodes. Therefore, we omit attribute and content nodes in this section for simplicity.

For representing open nodes separately from the close nodes in figures, we add one black dot: • to an open node representing the side on which the node is open, i.e. which tag is missing. For the example, when we split the pair of tags `<A></A>` into two tags, we can create an right-open node A• from `<A>` and a left-open node •A from `</A>`.

Last, based on the above definitions, we now give the definition to partial tree. A partial tree is a tree structure with open nodes and represents a chunk of an XML document and can be used for parallel XML processing.

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

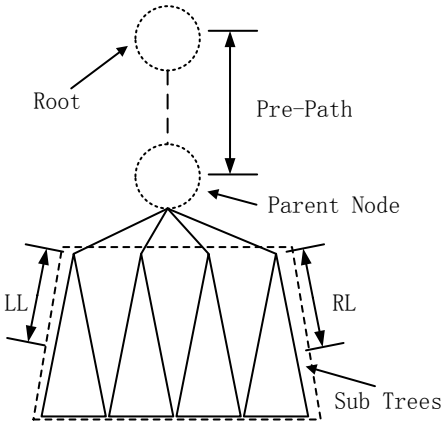


Figure 5.2: The standard structure of partial tree.

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 sibling relationship of the open nodes.

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

There is another important property of pre-nodes.

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

5.2.2 Standard Structure

According the properties of partial tree, the standard structure of partial tree can be described as in Fig. 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 emphpre-path. Pre-path plays an important role in applying queries from the root. From property 3, one or more subtrees connect to a pre-node at the bottom of the pre-path. Note that for each subtree, there is only one root, which is a left-open/closed/right-open node, but there could be one or more subtrees.

From properties 1 and 2, we know that 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, especially the pre-path, ordinary XML parsing algorithms, in turn, do not work for the construction of partial tree. The difference mainly lies in two aspects. First, an XML document can generate only a single XML tree. While in case of partial tree, the number of XML trees could be many, which is determined by the number of chunks. However, we cannot simply construct an partial tree from a chunk. This is caused by the second reason that 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 is the minimum subgraph. It should satisfy the following three conditions:

(1) the subgraph is connected (This means the subgraph is a tree.);

(2) each node in 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>
```

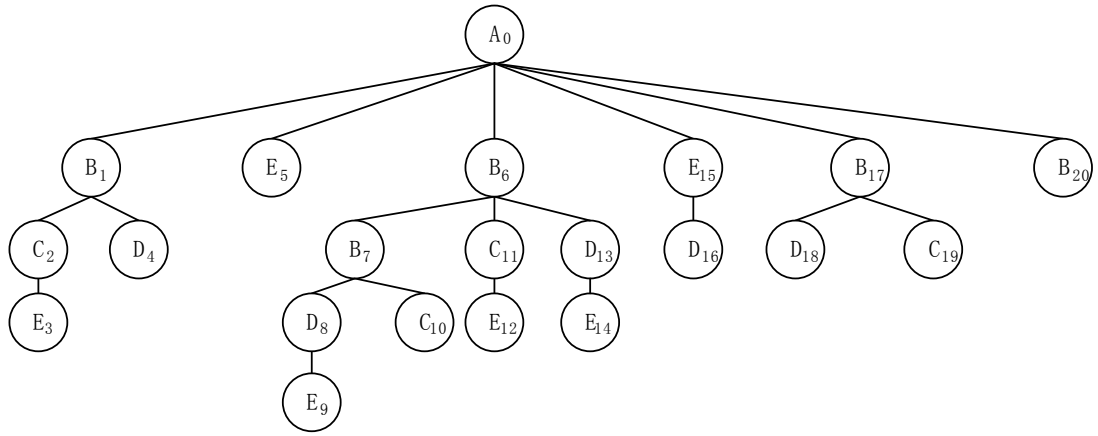


Figure 5.3: an XML tree from the given XML string

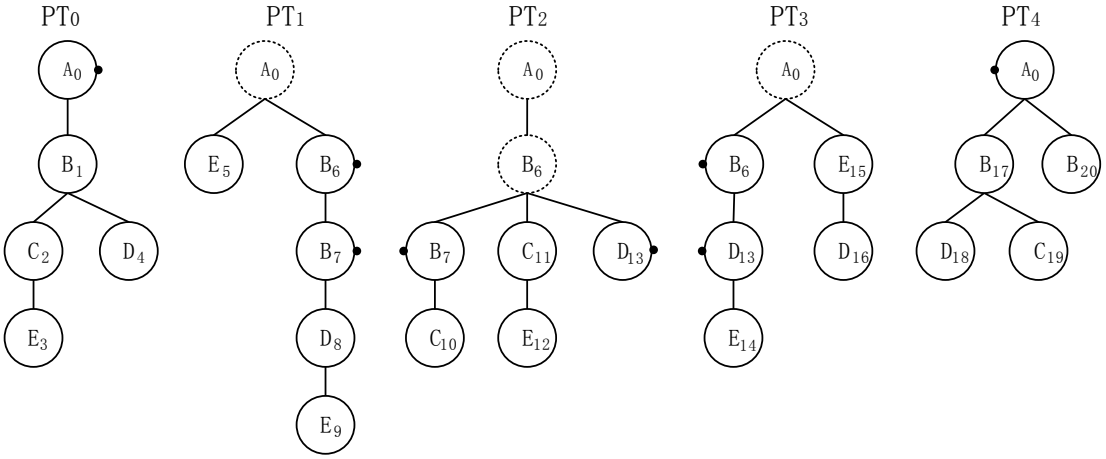


Figure 5.4: Partial trees from the given XML string.

From the document, we can construct an XML tree as shown in Fig. 5.3. We number all the nodes of the tree in a prefix order for identification.

To construct partial trees from the document, 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 a parital tree from each chunk, i.e. chunk$_i$ makes a partial tree

PT_i, as shown in Figure 5.4.

In the following section, we introduce our partial tree construction algorithm with the example. Our algorithm is a three-phase algorithm. In the first phase, we construct mutiple sets 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 open nodes. Last, we add pre-paths to the corresponding list of subtrees to complete the partial tree construction. We will give deailed introduction to the three-phase construction algorithm of partial tree. Furthermore, to server the query algorithm, we also introduce the statistics information of open nodes, called ranges of open nodes. With such information, we can easily access open nodes of the same node and synchronize the query results among partial trees.

5.3.1 Construction of Subtrees From Parsing XML Chunks

As introduced previously, a partial tree is constructed from parsing an input XML chunk, which is a substring generated from splitting the XML document. 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.

First, after splitting an XML document, we deal with nodes with missing tags. During parsing, 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. 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 Fig. 5.5.

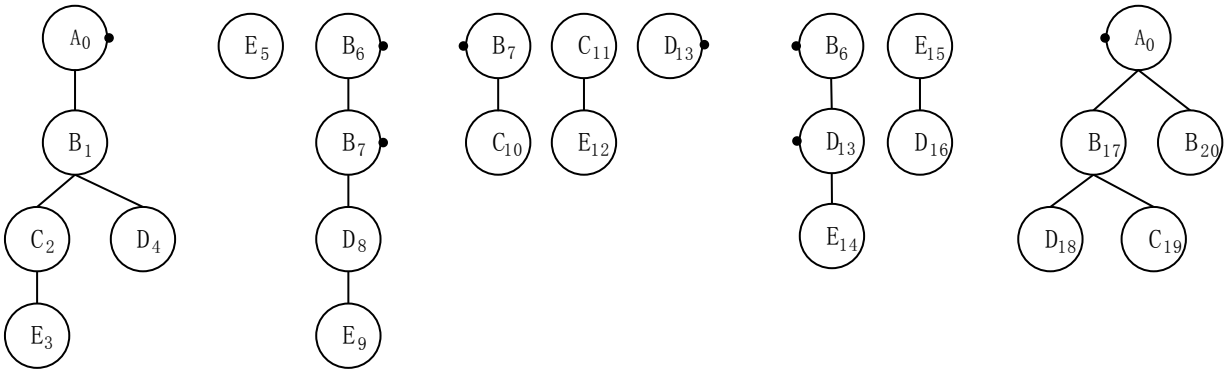


Figure 5.5: Subtrees from parsing chunks

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 those nodes. Therefore, the information needed for creating the pre-paths lies in them.

Algorithm 0 outlines the pseudo codes for computing pre-path. Since one chunk may generate more than one subtrees, the input is a list of subtree lists. The length of the list is equal to the number of partial trees, i.e. one chunk makes one list of subtree, thus representing the length as P.

Algorithm 0 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 pth 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 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 from a position inside the document, the two partial trees created from the splitting have the same number of open nodes on the splitting side. Given two consecutive partial trees, the number of the right-open nodes of the left partial tree is

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

the same as the number of the left-open nodes of the right partial tree. This is a very important feature and we exploit it for computing the pre-paths of partial trees.

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.

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

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

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 Fig. 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 distrubted 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 nodes to a partial tree, the open nodes from the same node also have the same depth. Therefore, we can locate all the matching nodes to set *start* and *end* for each open node. After computation, we obtain the ranges of nodes as shown in Table 5.4.

By using these ranges, we can esaily establish 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.

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)]	[]

5.4 Eavalute XPath Queries on Partial Trees

When performing an XPath query on an XML document, the evaluation is done on a single tree. By exploiting partial tree, the evaluation of an XPath query on a single tree is then applied to multiple partial trees. A significant advantagte of using partial tree is that it makes the parallelization of evaluation of XPath queries possible by simply evaluting the same query on partail trees separately. However, it also brings challenges in designing query algorhtms on partial trees. Generally speaking, there are three main difficulties as follows.

First, since partial trees are craeted from chunks of an XML document, a node may be seperated 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. Since these open nodes are from the same node, 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. This is simply because they are all $\texttt{B}_6$ in the original XML document.

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 B_1 because of splitting. When we perform queries with `following-sibling` 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 follwoing 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 Definitions

For introducing the query algorithms, we first give a few definitions to partial tree nodes. 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 pointers pointing to the related nodes: the *parent* pointer that points to its parent and the *children* pointer to its children. For accessing siblings, it has the *presib* pointer and the *folsib* pointer 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,

we number each partial tree with a unique id denoted as *partial tree id* or simply *ptid* for distinguishing partial trees. We number *ptid* from 0 to *P - 1* (where P is the total number of partial trees) in document order.

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.

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 reslultant lists given P partial trees. The evaluation of a step is applies to each node in the resultant list. To start with, we add a virtual node VN*i* into a resultant list for each parital tree, which has the root of PT*i* as its single child. For example, VN_0 has only a single child $A_0\bullet$ of pt_0 and is put into the resultant list before the evaluation on pt_0 starts. The evaluation of a step generats 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.

5.4.2 Queries without Predicate

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 matchs the XPath query, each of which is associated with a corresponding partial tree.

The evaluation of the XPath query starts from the root 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

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

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.

5.4.2.1 Downwards Axes

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

Algorithm 3 shows the procedure for evaluting 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. 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 process of downward steps of Q1 is listed in Table 5.5. For the first step `child::A`, since VN_i has only one child that

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

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

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

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

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

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

Figure 5.7: Query algorithm for downwards axes

is the root of the ith partial tree, we obtain it as the results 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 results of the first step. We evalute the step `descendant::B` on each of the nodes in the resultant list of the previous step. For example, for $\texttt{A}_1\bullet$ on pt_1, there is only a node $\texttt{B}_1$ that matches the query and is then selected and put into a new list. As introduced, when the evaluation is done, the new list replaces the current resultant list to be the resultant list up to the current step.

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`	$[C_2]$	$[\,]$	$[C_{10}, C_{11}]$	$[\,]$	$[C_{19}]$

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. 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 $B_6\bullet$, we traverse $B_7\bullet$, D_8, E_9, and then starting from $B_7\bullet$, we can stop the traversal immediately.

5.4.2.2 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 due to Property 1 in Section 5.2. Let an open node x be selected after a downwards query. Then, it should have started from an open node (this is an ancestor of x) and the corresponding nodes should have all been selected, which means all the nodes corresponding to x should be selected after the query.

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 is used for keeping the consistency of open nodes. It

consists of two parts[1].

First, 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. 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. 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, we can notice that $\texttt{B}_1$ is selected for it is the parent of $\texttt{C}_2$ on pt_0, while for $\bullet\texttt{B}_7$ and $\bullet\texttt{B}_6\bullet$, they are the parents of $\texttt{C}_{10}$ and $\texttt{C}_{11}$ respectively. The results are listed in the second row of Table5.6.

Here, unfortunately the node $\bullet\texttt{B}_7$ is selected on pt_2, but its corresponding node on pt_1, i.e. $\texttt{B}_7\bullet$, 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 $[\bullet\texttt{B}_7, \bullet\texttt{B}_6\bullet]$. Since they have ranges $(1, 2)$ and $(1, 3)$, respectively, pt_1 receives two nodes $\texttt{B}_7\bullet$ and $\texttt{B}_6\bullet$, pt_2 receives two nodes $\bullet\texttt{B}_7$ and $\bullet\texttt{B}_6\bullet$, and pt_3 receives one node $\bullet\texttt{B}_6$. By taking the union with the previous intermediate results, we obtain the final results as shown in the last row of Table5.6.

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 the table from evaluating the last step is then the final results of Q1.

5.4.2.3 Intra-sibling Axes

The `following-sibling` or `preceding-sibling` axes retrieve nodes from a set of nodes that are siblings 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 intro-sibling

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

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	$[B_1]$	$[B_6 \bullet, B_7 \bullet]$	$[\bullet B_7, \bullet B_6 \bullet]$	$[\bullet B_6]$	$[B_{17}]$

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. Algorithm 6 shows the procedure for evaluating a step with a following-sibling axis, which consists of four phases: local query, preparation, regrouping, and remote query.

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

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

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 simple steps that have no predicates. Our algorithm for handling predicates consists of three phases: preparing, evaluating steps in predicates, and processing predicates. The main

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`	[]	$[B_6\bullet]$	$[\bullet B_6\bullet]$	$[\bullet B_6]$	$[B_{17}, B_{20}]$

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

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

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

Figure 5.9: Algorithm for Following-sibling axis

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	$[B_1 \rightarrow \{pt_0.B_1\}]$	$[B_6\bullet \rightarrow \{pt_1.B_6\bullet\}, B_7\bullet \rightarrow \{pt_1.B_7\bullet\}]$	$[\bullet B_6\bullet \rightarrow \{pt_2.\bullet B_6\bullet\}, \bullet B_7 \rightarrow \{pt_2.\bullet B_7\}]$	$[\bullet B_6 \rightarrow \{pt_3.\bullet B_6\}]$	$[B_{17} \rightarrow \{pt_4.B_{17}\}, B_{20} \rightarrow \{pt_4.B_{20}\}]$

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 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 $x \rightarrow \{pt_i.y\}$ in the running example, both of which mean node x is selected and it has a link to node y on pt_i.

5.4.3.1 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.

5.4.3.2 Evaluation of Steps within A Predicate

The evaluation of inner steps of a prediate is almost the same as that without predicate. Algorithm 9 shows the procedure for evalauting a 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 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 prediate 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 that is 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, $\texttt{B}_{17}$ and $\texttt{B}_{20}$ 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 $\texttt{B}_{20}$ in pt_4. The local result of it is $\texttt{B}_{20} \rightarrow \{\text{pt}_4.\texttt{B}_{20}\}$, and the remote results is $\texttt{B}_{20} \rightarrow \{\text{pt}_0.\texttt{B}_1, \text{pt}_3.\bullet\texttt{B}_6\}$. After merging link, we have $\texttt{B}_{20} \rightarrow \{\text{pt}_0.\texttt{B}_1, \text{pt}_3.\bullet\texttt{B}_6, \text{pt}_4.\texttt{B}_{20}\}$ 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 $[\texttt{C}_{19} \rightarrow \{\text{pt}_0.\texttt{B}_1, \text{pt}_3.\bullet\texttt{B}_6\}]$ is a child of $\texttt{B}_{17}$. Thus, it follows the link of $\texttt{B}_{17}$.

5.4.3.3 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

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`	[]	[]	$[\bullet C_{11} \rightarrow \{pt_0.B_1\}]$	[]	$[C_{19} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}]$

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`	$[C_2]$	[]	$[C_{11}]$	[]	[]

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. Links $C_{11} \rightarrow \{pt_0.B_1\}$ in the intermediate results of pt_2 adds node B_1 to the result list of pt_0 and $C_{19} \rightarrow \{pt_0.B_1, pt_3.\bullet B_6\}$ 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:

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

Input: $InputList_{[P]}$: an indexed set of lists of nodes
Output: an indexed set of lists of (node, link)
1: **for** $i \in [0, P)$ **do**
2: $\quad OutputList_p \leftarrow [(n, (p, n.uid))|n \in InputList_p]$
3: **return** *OutputList*

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

Input: $pt_{[P]}$: an indexed set of partial trees
$\quad InputList_{[P]}$: an indexed set of lists of (node, link)
Output: an indexed set of lists of filtered nodes
1: /* regroup links by partial tree id. */
2: $AllLinks \leftarrow [\,]$
3: **for** $i \in [0, P)$ **do**
4: $\quad AllLinks \leftarrow AllLinks \cup [(p', i')|(n', (p', i')) \in InputList_p]$
5: **for** $i \in [0, P)$ **do**
6: $\quad Activated_p \leftarrow [n \mid (p', i') \in AllLinks, p = p', n.uid = i']$
7: **return** SHARENODES($pt_{[P]}$, $Activated_{[P]}$)

Figure 5.10: Query algorithm for handling predicate

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

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

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 5.11: Query algorithm for child axis in a predicate

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

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			$O(P^2K^2)$	$O(P^2K^2)$

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.

Indexing is a common used technique in database community for efficient aceess of data [**?** **?**]. Agarwal et al [**?**] proposed an idea for processing queries on compresssed XML data for memory efficiency. While Krulis et al [**?**] studied how to process queries on multi-core. In this section, we propose an effecient implementation of partial tree based on two indexes: BFS-array index and grouped index in considering the characteristic of partial tree. 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 as an example, an XML tree can be constructed from the XML document as shown in Figure 5.12.

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

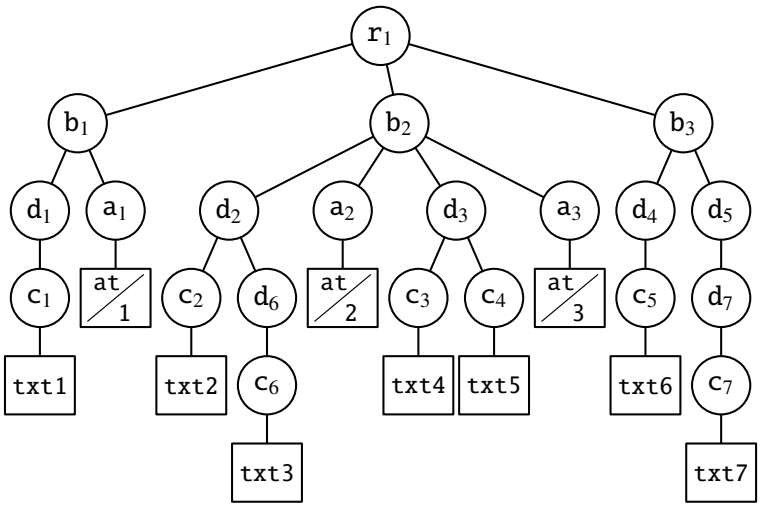


Figure 5.12: An example XML tree with values.

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, there is no difference in applying partial tree. We then can construct four partial trees as shown in Figure 5.13.

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 implementated by the idea of partial tree. As we have introduced 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.

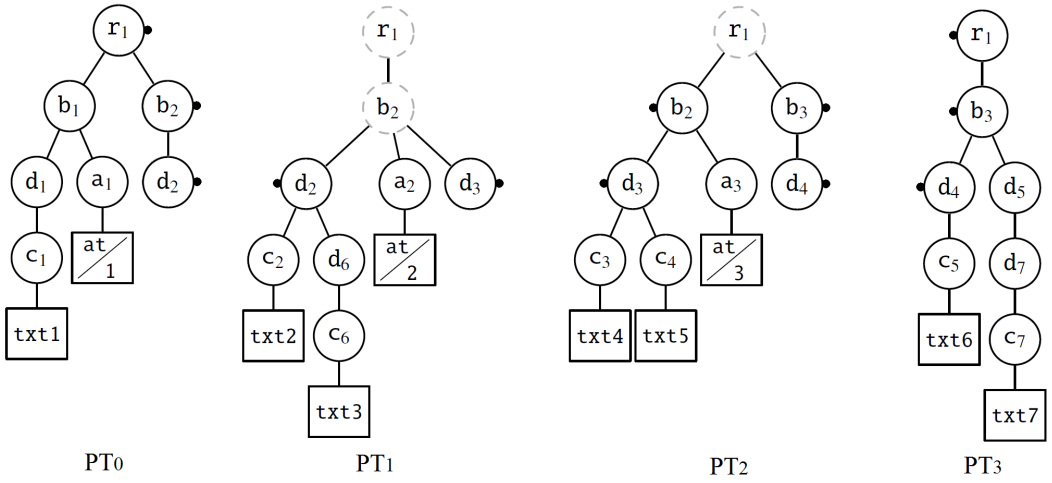


Figure 5.13: Partial trees with values from the XML document.

Expressiveness

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

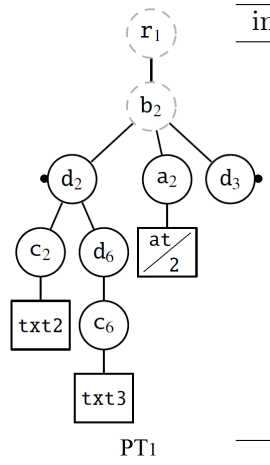
Compactness

In the case we repeatedly apply several queries on the same data, we can put all the indices 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 [**?**]. However, such an index with the update capability tends to be large and complicated to handle. 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.

Partial tree



BFS-array

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 5.14: A partial tree and its representation with two arrays

- 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.
- Function getNodesIn(t, x) returns all the nodes with tag t in the subtree rooted at x.

We design two index sets (Fig. 5.14) 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 explore the query performance of our implementation on processing in two cases: (1) we use a single EC2 instance with a single partial tree to investigate the query performance of partial tree in serial; (2) we use 32 EC2 instances to show the query performance for processing very large XML documents in parallel.

5.6.1 Datasets and XPath Queries

Table 5.12 shows the statistics of XML data sets. For the experiments on a single EC2 instance, we used two typical datsets: DBLP and XMark [**?**] (with factor 100). For parallel processing, we used XMark(with factor 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 2 shows the 15 queries covering the three cases: XQ1 and UQ1 to test long queries with nested predicates; XQ2, DQ1, DQ2, UQ2, UQ4 and UQ5 to test backward axes; and the rest to test order-aware queries.

5.6.2 Experimental set-up

We used single or multiple m3.2xlarge instances on Amazon EC2. The m3.2xlarge instances are based on E5-2670 v2 (Ivy Bridge), equipped with 30 GB of memory and 2 X 80 GB of SSD, running Amazon Linux AMI 2016.09.0. Our prototype is implemented in Java 1.6 and executed on 64-Bit JVM (build 25.91-b14).

5.6.3 Evaluate Queries on a Single EC2 Instance

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

5.6.4 Evaluate Queries on Multiple EC2 Instances

In this experiment, we investigate the query performance processing very large XML document using multiple EC2 instances. We use UniProtKB and XMark(with factor 2000) as experimental data. The results are shown in Table 5.15. 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

Table 5.12: Statistics of XML dataset.

Datasets	dblp.xml	xm100.xml	xm2000.xml	uniprot.xml
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 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	xmark10.xml					dblp.xml				
Time	8.5					47				
Memory	222					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.xml					unirpot.xml				
Loading (ms)	210					379				
Memory (GB)	173					560				
Query	QX1	XQ2	QX3	QX4	QX5	UX1	UX2	UX3	UX4	UX5
Time Taken (ms)	5951	819	1710	1168	3349	2573	2408	1324	5909	6220

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 [**?**] 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.

5.7 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 with two approaches, BaseX and 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 queires 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 databasex 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 subqueries 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 subqueries 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 evalaution of XPath queries over gigabytes XML documents, which is very convinient and practical.

For processing larger XML documents, 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 size so that we can reach good load-balancing. Second, partial trees are portable in both shared-/distributed- memory environemts. This means that we can make them work in both distributed-memory environments and shared-memory 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 three works that are worthing doing in the future.

Firstly, our implementations on top of BaseX were evaluated with only a single

BaseX server on a dual-CPU system. Thus, it would be very promising to use multiple BaseX servers on multiple CPUs in distributed-memory environments over large XML documents. Considering the PRE values that are integers and very small, it is suitable to represent the intermediate results of queries by PRE values to be transmitted among BaseX processors. With more BaseX servers running on a computer cluster, it is feasible to efficiently process larger XML documents.

Secondly, although partial trees are suitable for processing XPath queires 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.

Lastly, 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.

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other Japan or foreign examination board.

The thesis work was conducted from Wei HAO under the supervision of Associate Professor Kiminori Matsuzaki at Kochi University of Technology, Kami City, Kochi Prefecture, Japan.

Signature:

Date:

Bibliography

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other Japan or foreign examination board.

The thesis work was conducted from Wei HAO under the supervision of Associate Professor Kiminori Matsuzaki at Kochi University of Technology, Kami City, Kochi Prefecture, Japan.

Signature:

Date: