A Dual-Index Based Representation for Processing XPath Queries on Very Large XML Documents

Wei Hao[1,2(✉)], Kiminori Matsuzaki[2], and Shigeyuki Sato[2]

[1] Anhui University of Science and Technology, Taifeng Avenue 168, Huanian, Anhui, China
whao@aust.edu.cn

[2] Kochi University of Technology, 185 Miyanokuchi, Tosayamada, Kami, Kochi 782–8502, Japan
{matsuzaki.kiminori,sato.shigeyuki}@kochi-tech.ac.jp

Abstract. Although XML processing has been intensively studied in recent years, designing efficient implementations for evaluating XPath queries on XML documents remains a challenge in case XML documents are very large. In this study, we implemented a tree-shaped data structure called partial tree that is intrinsically suitable for large XML document processing with multiple computers. Our implementation uses two index sets to accelerate the evaluation of structural relationships among nodes, making it highly efficient for processing very large XML documents regarding three important classes of XPath queries: backward, order-aware and predicate-containing queries. Experiment results show that our implementation outperforms a start-of-the-art XML database BaseX in both absolute loading time and execution time for the target queries. The absolute execution time over 358 GB of XML data averagely is only seconds by using 32 EC2 instances.

Keywords: Large XML documents · XPath querying · Dual-index · Data representation · Parallel computing

1 Introduction

XML is a popular data representation, storing and exchanging language widely used in many areas, such as in database systems and web services. XPath[1] and XQuery[2] are used for navigating through elements of XML documents. The execution time of these queries come to matter in processing of very large XML documents. Even though common query processors on commodity computers suffice for evaluating queries over commonplace documents, they do not suffice

[1] https://www.w3.org/TR/xpath/.
[2] https://www.w3.org/TR/xquery/.

L. Qi et al. (Eds.): CloudComp 2020, LNICST 363, pp. 18–30, 2021.
https://doi.org/10.1007/978-3-030-69992-5_2

for doing over very large XML documents such as a DBLP[3] document dump of 1.8 GB and an UniProtKB[4] document of 358 GB. The current XML processing techniques enable us to evaluate some queries efficiently. For example, a technique [14] for extracting parent-child and ancestor-descendant relationships is efficient because it uses indices regardless of the order of results. However, due to the intrinsic nature that the elements of an XML document are ordered, order-aware queries that contain axes such as `following`, `following-sibling`, `preceding`, `preceding-sibling` axes and `position` function are important. Backward queries such as `parent` and `ancestor` axes used for tracking back toward the root of the XML documents are also important. Some algorithms such as structural join [1] use a join algorithm on two lists (AList for ancestors and DList for descendants) to determine child-parent and ancestor-descendant relationships. However, it takes $O((|AList| + |DList|)^2)$ time in the worst case. Thus, it will take terrible time in processing very large documents. Besides, predicates, which are an construct for filtering elements, are useful for queries, while queries efficient evaluation of predicate-containing queries also becomes an issue.

In this study, we aim at good absolute time, which is the shortest time for processing a certain amount of data, of evaluating over large XML documents, XPath queries including three classes: backward, order-aware and predicate-containing queries. We propose a novel compact representation for partial trees [11], which is a data structure dedicated to processing large XML documents on multiple computers. This representation consists of two index sets: one accelerates extracting relationships between nodes; the other enables us to avoid evaluating uninvolved nodes. With this two index sets, we can process very large XML documents efficiently. Our contributions are summarized as the follows:

- We have proposed a compact tree representation that can be used for efficiently process the three classes of XPath queries.
- Our prototype implementation have outperformed BaseX[5] and achieved good absolute evaluation time.

The experiments were conducted on Amazon Elastic Compute Cloud[6] (EC2) cloud server, which is a web service and provides flexible compute capacity in the cloud. The experiment results show that our approach outperforms a start-of-the-art XML database BaseX in better absolute time. The execution time over 358 GB of XML data is only seconds averagely.

Figure 1 shows the syntax of our target subset of XPath. Our prototype implementation accepts nested predicates, which means the location steps inside a predicate can still have predicates, making queries in this study more expressive. For the `position()` function, only `child` axis is supported.

The rest of this paper is organized in the following: Section 2 introduces our previous work: partial trees. Section 3 presents the novel compact representation.

[3] http://dblp.uni-trier.de/.
[4] http://www.uniprot.org/help/uniprotkb.
[5] http://basex.org/.
[6] https://aws.amazon.com/ec2/.

Section 4 shows the querying algorithms used in the experiments. Section 5 reports results and analysis. Section 6 concludes the paper and show the future work.

Query ::= '/' *LocationPath*
LocationPath ::= *Step* | *Step* '/' *LocationPath*
Step ::= *NameTest Predicate?* | *NameTest* [*integer*]
| '/'*NameTest Predicate?* | *Axis::NameTest Predicate?*
AxisName ::= 'child' | 'parent' | 'descendant' | 'ancestor'
| 'descendant-or-self' | 'ancestor-or-self' | 'following'
| 'following-sibling' | 'preceding | 'preceding-sibling'
NameTest ::= '*' | *string*
Predicate ::= '[' (*LocationPath* | *position*() = *integer*) ']'

Fig. 1. Grammar of our target subset of XPath.

2 Related Work

Existing studies [5,7,8] on evaluating XPath queries over distributed (or fragmented) XML documents dealt only with top-down path queries (also known as twig patterns), which compose a small subset of navigational XPath queries. To the best of our knowledge, navigational XPath queries over distributed documents have never been seriously investigated.

It is worth noting that applying MapReduce [9] to large-scale XML processing was well studied [5,6,8,15,16]. MapReduce enables transparent and fault-tolerant processing on clusters, while encoding XPath queries on it incurs overheads in various aspects. In order to minimize overheads and achieve high performance on clusters, dedicated in-memory processing is more appropriate than MapReduce-based processing. VXQuery [4] has been designed without MapReduce for this reason but still been under development. At least, how to process navigational XPath queries in VXQuery was not described in [4].

Apart from parallel and/or distributed processing, efficient evaluation of more expressive subsets of XPath was studied [2,3,10]. Brantner et al. [3] studied efficient compilation of XPath 1.0 with the internal algebra of an underlying XML database engine. Grust [10] accelerated evaluation of all axes with an indexing technique. SXSI [2] dealt with forward Core XPath and all text predicates in a succinct index structure. We similarly have used two indices represented with a dedicated data structure for accelerating query evaluation. We, however, do not claim that these indices and representation are technically better than existing ones. We have experimentally demonstrated that the two cheap indices in the straightforward representation based on partial trees suffice for achieving high performance processing of navigational XPath queries over large-scale documents, not only on a single computer but also clusters—this is our technical contribution.

3 Partial Tree

For high-performance data processing, a common idea is to divide a large amount of data into smaller fragments and distribute them to multiple processors for later processing. Consider an XML tree in Fig. 2 and its serialized document is divided into the following chunks ($chunk_0$ to $chunk_3$ in document order). When an XML document is divided, we could parse a chunk to construct a subgraph (a sequence of subtrees). For example, given $chunk_2$, we can construct a subgraph as shown in Fig. 3. However, because of the intrinsic tree structure of XML documents, queries cannot be directly evaluated on them after splitting. For example, we cannot select the children of b2 on the subgraph from $chunk_2$, because the path from the root of the subgraph to the root of the whole tree is missing.

$chunk_0$: `<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>`

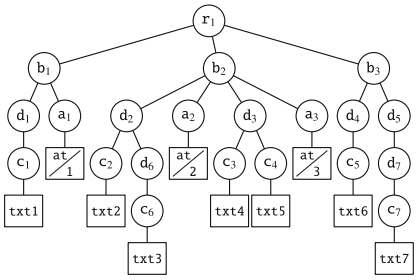


Fig. 2. An example XML tree.

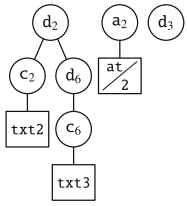


Fig. 3. A subgraph from $chunk_2$.

To cope with the path-missing issue, we exploit a data structure called *partial tree* [11] that is suitable for representing chunked XML data of an XML documents. A partial tree can be constructed in two steps: we first parse a chunk of an XML document to generate a sequence of subtrees, and then add the path for each root of subtrees to the root of the original XML tree. For example, four partial trees are constructed from parsing $chunk_0$ to $chunk_3$ as shown in Fig. 4. A partial tree corresponding to a chunk is the minimum subgraph, that satisfies three conditions:

- a subgraph is connected, which means a subgraph is a tree.
- a subgraph contains the root of the original XML tree.
- elements(tag/attribute/text) from the same chunk are in the same subgraph.

A concrete algorithm for this computation is given in [11] (Algorithm 0). In this study, the chunks of a divided XML document begin at either a start tag or an end tag to keep the attributes and texts as closed nodes. It is useful to distinguish the four cases that a node in a partial tree has or does not have its tags in the corresponding chunk. Nodes in a partial tree are categorized into the following four types (in Fig. 5) based on the inclusion of tags in the chunk: a *closed node* with both its tags, a *left-open node* with only its end tag, a *right-open node* with only its start tag, and a *pre-node* with no tags. The left-open nodes, right-open nodes and pre-nodes are called *open nodes*.

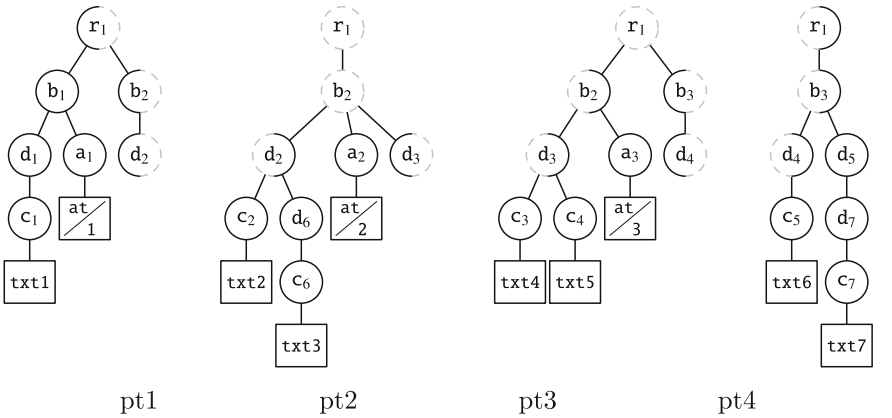


Fig. 4. Four partial trees constructed from chunks.

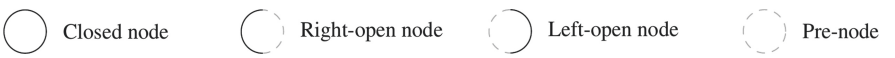


Fig. 5. Node types.

4 Data Representation

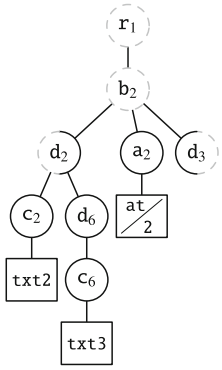
From the previous section, we introduced a fragmentation based on a data structured called partial trees to represent a chunked XML document, providing a good way for parallel XML processing. However, it is still a challenge to develop a high-performance implementation based this data structure, especially the concrete representation for it. For this purpose, we take two issues into consideration as the following:

Expressiveness. We use indexing (or labeling) to represent chunked XML data so that it can be efficient to store them in databases [13], exploiting its expressiveness to accelerate queries.

Compactness. The indices for the XML data are stored in memory, so that the expensive I/O cost for data exchanging are avoided.

The first design requirement relates to the efficiency of XML update. It is common that a general-purpose framework may provide supports for updates, such as ORDPATH [13]. However, such configuration tends to be expensive in case of query-intensive(or only) scenario. Therefore, we focus only on update-free case, i.e. we consider only queries with no update, making our framework much faster and easier to implement. We expect that users are able to achieve their objectives without updating, as long as appropriate programming interfaces over the framework are provided.

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

Grouped-array

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

Fig. 6. A partial tree and its representation with two arrays

We consider the second requirement as the functionality our designed indices can provide. One goal of this study is to support queries with order-aware ones such as `following-sibling` and `following`. To achieve our goal, we define the following functions for processing queries on the given node x:

- Function getChildren(x) returns the children of x.
- Function getParent(x) returns the parent of x.
- Function nextSibling(x) returns the next sibling of x.
- Function prevSibling(x) returns the previous sibling of x.
- Function isDescendant(x, y) returns true if x is a descendant of the node y.
- Function isFollowing(x, y) returns true if x is strictly after the node y in terms of the document order.
- Function getNodesIn(t, x) returns nodes with the tag t in the subtree rooted at x.

For the implementation of the above functions, we designed two compact index sets, BFS-array (BFS stands for Breadth First Search, since nodes are arranged in BFS order) and Grouped-array, as shown in Fig. 6. For a single node, we design the following properties:

- *tag*: the tag name of the node, short integers that map to the strings.
- *type*: the type of the node, including four node types as shown in Fig. 5.
- *st*: the start position of the node, which is the position of the tag in the original file to avoid global counting.
- *ed*: the end position.

With BFS and these two pointers, we are able to perform high-performance functions getChildren, getParent, nextSibling, and prevSibling. With *Grouped-array*, we are allowed to evaluate the function getNodesIn efficiently.

In this implementation, the two indexes totally take 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*. The sum of the above is merely 31 bytes for representing an XML node. To the best of our knowledge, it is smaller than the known existing implementations of DOM trees or XML databases, and it is also the key to high-performance queries executions.

5 Query Algorithms

By dividing an XML document into chunks, multiple partial trees are constructed. The evaluation of queries applied to the original XML document, can be are applied to partial trees and the evaluation on theses partial trees can be done separately. The Overall Algorithm outlines the "big picture" of evaluating a query on multiple partial trees. The query starts from the root of the XML tree. Note that the root node corresponds to the root node of every partial tree, and they are put into the lists for intermediate results (lines 1–2). Hereafter, the loops by p over $[0, P)$ are assumed to be executed in parallel. An XPath query consists of one or more steps, and in our algorithm they are processed one by one. For each step, our algorithm calls a sub-algorithm based on its axis (given later) and updates the intermediate results (line 4). Lines 6–9 will be executed when a query has a predicate (Fig. 7).

The number of partial trees is the same as the number of the chunks. In case the number of partial trees is one, the whole original XML document is treated as a chunk, so that only one partial tree is constructed to work standalone without needing to make any change to its configuration. This is a valuable property of partial tree, making partial tree easily portable under different hardware settings.

We use the two index sets to implement the functions in Sect. 3. For `child` axis, getChild(x) is implemented by a set of indices as $[ch[x], ch[x+1])$. For example, in Fig. 6, the index of $\bullet\mathsf{B}_6\bullet$ is 2 and its children are in the range $\mathsf{getChild}(2) = [ch[2], ch[2+1]) = [3, 6)$. For `descendant` axis, isDescendant(x, y) can be implemented by $st[y] < st[x]$ *and* $ed[x] < ed[y]$. For `parent` axis,

Overall Algorithm QUERY($steps$, $pt_{[P]}$)

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

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

Fig. 7. The overall algorithm for processing XPath queries over partial trees

getParent(x) is implemented by $par[x]$. For example, the index of •B$_6$•'s parent is $par[2] = 1$. Since a node in the original XML tree may be split into two or more nodes on different partial trees. When such a node is selected in a partial tree (e.g., B$_6$• on pt$_1$), the other corresponding nodes (•B$_6$• on pt$_2$ and •B$_6$ on pt$_3$) should also be selected to be consistent. When an open node in the results is needed to share, we simply use *start* as index to notify other partial trees and let them to put the open nodes with the same *start* into the results nodes. We call this process share nodes. After processing `parent` axis, we need share nodes. For `ancestor` axis, we implement it by repeated proceeding from child to parent.

Without loss of generality, the discussion focuses on `following` and `following-sibling` axes only (`preceding` and `preceding-sibling` axes are just in opposite direction). For `following` axis, it is relatively easy that we simply compare the *end* index. We first select the node with the smallest *end* index. Then, nodes with *end* index greater than the smallest one and match the name test are selected as the results. For `following-sibling`, we need a two-phases query: the local query phase and the remote query phase. In the local query phase, we utilize the *folsib* to select siblings on the local partial tree. Then, we will ask a remote query if the parent node can have more segments on the right (i.e., right open). In the remote query phase, we select children and on the remote partial trees and merge them with their local results as final results. Note that after processing these axes, sharing nodes is needed.

We exploit the algorithms proposed in [11] (Algorithm 7 and 8) for predicate. The key idea of these two algorithms are introduced as follows: (1) create links that point to the nodes in the input lists, (2) evaluate the steps within the predicate with these links, (3) after the evaluation of all steps, we use the links to track back to the nodes in the input lists as final results.

6 Evaluation

Our evaluation have two aims: (1) to demonstrate the performance of our implementation against the state-of-the-art XML database BaseX on a single computer and (2) to explore the scalability of our implementation for processing very large XML documents on multiple computers in a parallel manner.

Datasets and XPath Queries. Table 1 shows the statistics of the XML datasets used in the experiments. For the experiments using a single EC2 instance, two XML datasets are used: DBLP and xmark100 (with factor 100[7]). For the experiments using mutiple EC2 instances, xmark2000 (with factor 2000) and UniProtKB is are used. The UniProtKB dataset is well-balanced and has has a large number of children to its root, while XMark datasets are not well-balanced for they have only six children to their root. Table 2 shows the queries: XQ1 and UQ1 for long queries with nested predicates; XQ2, DQ1, DQ2, UQ2, UQ4 and UQ5 for queries with backward axes; the rest is for order-aware queries.

Table 1. Statistics of XML dataset.

Datasets	dblp	xmark100	xmark2000	uniprot
Nodes	43.13M	163.1M	3.26B	7.89B
Attributes	10.89M	42.26M	845M	9.25B
Values	39.64M	67.25M	1.34B	1.49B
Total	93.66M	272,67M	5.45B	18.64B
# of distinct tags	47	77	77	82
Depth	6	13	13	7
File size (GB)	1.78	10.95	220	358

Experiment Configuration. There were 32 m3.2xlarge instances on Amazon EC2, which were equipped with E5-2670 v2 (Ivy Bridge), 30 GB of memory and 2 X 80 GB of SSD, running Amazon Linux AMI 2016.09.0. Our prototype was implemented in Java 1.6, running on 64-Bit JVM (build 25.91-b14).

Comparison with BaseX. To compare with, we selected BaseX 8.5.3 (released on August 15, 2016) as our opponent. It was tested under two configurations: one was to put all the XML data and index sets on memory (*BXon*), and the other was to put only the index sets on memory (*BXoff*). In order to eliminate

[7] The factor determines the file size of an XMark generated document. It is nearly linear: 1 = 110 MB, for example xmark100 with the factor 100 is about 11 GB, while xmark2000 with the factor 2000 sized 220 GB.

Table 2. 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

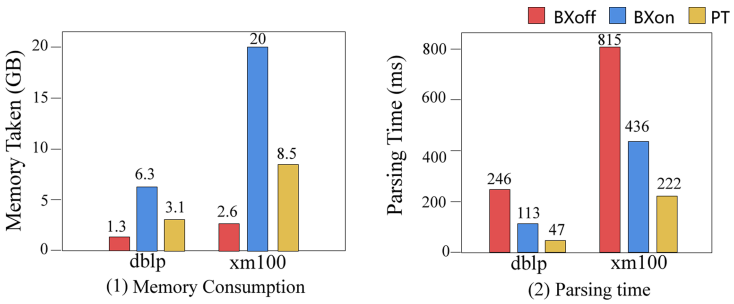


Fig. 8. Memory consumption and parsing time on dblp datasets

Table 3. Evaluation by 32 EC2 instances

Dataset	xm2000					unirpot				
Loading (s)	210					379				
Memory (GB)	173					560				
Query	QX1	XQ2	QX3	QX4	QX5	UX1	UX2	UX3	UX4	UX5
Time (ms)	5,951	819	1,710	1,168	3,349	2,573	2,408	1,324	5,909	6,220

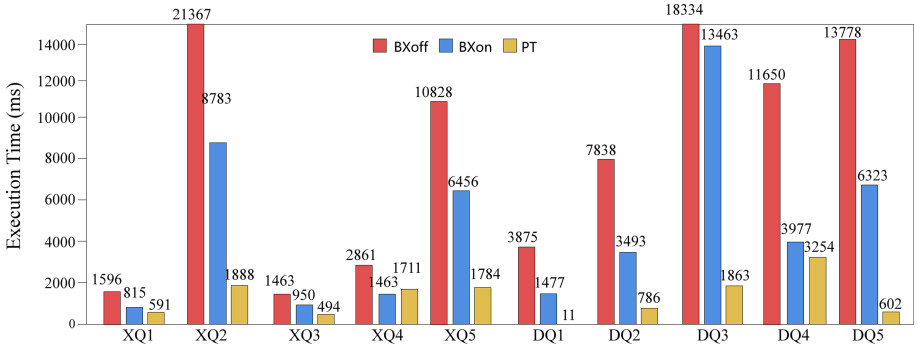


Fig. 9. Execution time of queries on XMark100) and dblp datasets

the influence of printing time for output, we simply apply count function to queries, e.g.. "count (XQ1)"; then use the time for evaluating this query as execution time. We also set the parameter INTPARSE true to use the internal XML parser that is faster, more fault tolerant and supports common HTML entities out-of-the-box.

Evaluating Queries on 1 EC2 Instance. We first conducted the experiment on 1 EC2 instance to investigate the querying performance in comparison with BaseX using the queries listed in Table 1 on XMark (with factor 100) and dblp. The experimental results as shown in Fig. 9 clearly demonstrates that our approach outperforms BXon and BXoff in most tests except for XQ4. In most case, our implementation achieves 2-6 times faster than BaseX. In some extreme case, we can achieve 100s times faster than BaseX (DQ1). The reason in this case is because we group nodes with the same tag name, avoiding evaluating unnecessary nodes and the parent-child relationship can be determined in constant time. We also notice that BXon is 2–3 times faster than BXoff. We believe this is simply because it loads all data into memory. This can be learnt from Fig. 8(1). It also shows that Bxon takes more memory than ours for all the datasets. Even we deduct the size of original XML datasets, it still exceeds our approach. As of the parsing time, we can see that in Fig. 8(2). Our approach is nearly twice faster than BXon and four times faster than BXoff in both datasets.

Evaluating Queries on 32 EC2 Instances. In this experiment, we investigated the querying performance on processing very large XML documents by using 32 EC2 instances. We used UniProtKB and XMark (with factor 2000) as experiment data. The results are shown in Table 3. With the design of 31 bytes for a node, the memory consumption should be 157 GB and 537 GB for 0.545 billion and 1.86 billion nodes respectively. The experiment results show the consumptions are 173 and 560 GB, much close to the computation. We believe the overheads result in some intermediate data generated during the parsing phase.

The parsing times in Table 3 are relatively short considering the very large data sizes. The querying is also very fast, such as XQ1 to XQ5 that took just a few seconds. The throughput of most queries is about 1 GB/s. The best throughput of One study, PP-Transducer [12], achieved the throughput of 2.5 GB/s at most with 64 cores. Although it is faster, the queries we can process are more expressive than that, which does not support order-aware queries.

7 Conclusion

In this paper, we proposed a novel dual-index (BFS and GFS) based data representation based on our previous work partial tree for high-performance XML processing, especially in terms of three classes of navigational XPath queries over very large XML documents. The basis of this approach is the chunk-based fragmentation and querying algorithms that exploit in-memory representation of chunks by dividing an XML document. We also conducted several experiments on EC2 instances. The experiment results clearly showed the efficiency of our framework outperform the state-of-the-art XML processing engine BaseX on 1 EC2 instance and achieved the throughput of about 1 GB/s using 32 EC2 instances.

The current work has a configuration that all indices are storied in memory. Therefore, our future work is to extend the ability of our implementation by developing new mechanisms to cooperate with modern distributed file systems, such as HDFS with MapReduce and related computing frameworks. In more details, we will pay our concern on data feeding, more expressive querying algorithms, error tolerance and etc.

References

1. Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 12th International Conference on Data Engineering, pp. 141–152 (2002)
2. Arroyuelo, D., et al.: Fast in-memory XPath search using compressed indexes. Softw. Pract. Exp. **45**(3), 399–434 (2015)
3. Brantner, M., Helmer, S., Kanne, C.C., Moerkotte, G.: Full-fledged algebraic XPath processing in Natix. In: Proceedings of the 21st International Conference on Data Engineering (ICDE 2005), pp. 705–716 (2005)
4. Carman, E.P., Westmann, T., Borkar, V.R., Carey, M.J., Tsotras, V.J.: A scalable parallel XQuery processor. In: Proceedings of 2015 IEEE International Conference on Big Data, pp. 164–173 (2015)
5. Choi, H., Lee, K.H., Kim, S.H., Lee, Y.J., Moon, B.: HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012), pp. 2737–2739 (2012)
6. Choi, H., Lee, K.-H., Lee, Y.-J.: Parallel labeling of massive XML data with MapReduce. J. Supercomputing **67**(2), 408–437 (2013). https://doi.org/10.1007/s11227-013-1008-6

7. Cong, G., Fan, W., Kementsietsidis, A., Li, J., Liu, X.: Partial evaluation for distributed XPath query processing and beyond. ACM Trans. Database Syst. **37**(4), 32:1–32:43 (2012)
8. Damigos, M., Gergatsoulis, M., Plitsos, S.: Distributed processing of XPath queries using MapReduce. In: Proceedings of the 17th East European Conference on Advances in Databases and Information Systems (ADBIS 2013), Part II, pp. 69–77 (2013)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
10. Grust, T.: Accelerating XPath location steps. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), pp. 109–120 (2002)
11. Hao, W., Matsuzaki, K.: A partial-tree-based approach for XPath query on large XML trees. J. Inf. Process. **24**(2), 425–438 (2016)
12. Ogden, P., Thomas, D., Pietzuch, P.: Scalable XML query processing using parallel pushdown transducers. Proc. VLDB Endow. **6**(14), 1738–1749 (2013)
13. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: insert-friendly XML node labels. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004), pp. 903–908 (2004)
14. Qin, L., Yu, J.X., Ding, B.: TwigList: make twig pattern matching fast. In: the 12th International Conference on Database Systems for Advanced Applications, pp. 850–862 (2007)
15. Sauer, C., Bächle, S., Härder, T.: Versatile XQuery processing in MapReduce. In: Catania, B., Guerrini, G., Pokorný, J. (eds.) ADBIS 2013. LNCS, vol. 8133, pp. 204–217. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40683-6_16
16. Wu, H.: Parallelizing structural joins to process queries over big XML data using MapReduce. In: Decker, H., Lhotská, L., Link, S., Spies, M., Wagner, R.R. (eds.) DEXA 2014. LNCS, vol. 8645, pp. 183–190. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10085-2_16