Cluster Computing (2024) 27:1375–1399
https://doi.org/10.1007/s10586-023-04010-0

Integrated method for distributed processing of large XML data

Rongxin Chen[1,2] · Guorong Cai[1] · Jie Chen[1] · Yuling Hong[1]

Received: 20 January 2023 / Revised: 18 March 2023 / Accepted: 15 April 2023 / Published online: 13 May 2023

Abstract
The traditional standalone computing approach is difficult to handle the task of processing large XML data due to scalability, thus distributed processing using cluster systems becomes an inevitable choice. The currently distributed XML processing methods generally rely on existing distributed computing frameworks for general purpose data, which have limitations such as complex configuration, inflexible working mechanism, and difficult performance optimization in the context of XML semi-structural features and complex queries. In addition, XML distributed queries suffer from a low level of automatic processing and lack of effective integration with distributed XML parsing and indexing. In this paper we propose an integrated method for distributed processing of large XML data, called the dXML method. Our method supports the distributed parsing of arbitrary XML fragment and the distributed creation of index, and adopts the efficient navigational XPath evaluation based on relation index. Through a distributed XPath evaluation approach based on filter-upon-pre-evaluate, our method enables data locality and reduces network traffic during the distributed evaluation of complex XPath predicates. dXML integrates the distributed processing technology of XML parsing, index creation and XPath query, provides a one-stop XML processing solution, supports the automatic distributed processing of large XML data, and has the characteristics of lightweight configuration and flexible working mechanism. Experimental evaluation verifies the effectiveness of dXML, and comparative experimental results show that dXML has better distributed query performance than both the typical existing navigational and Twig distributed processing methods.

Keywords Large XML data · XML parsing · XPath evaluation · Distributed processing · Integrated method

1 Introduction

With the popularity of the Internet and the rapid development of Web services, XML is widely used as a standard for information exchange and storage, and the technologies related to XML data processing are continuously developed [1, 2]. As the size of XML data generated in various applications is becoming larger and larger, it is difficult for a single computer to process it. For example, DBLP [3] is a computer science bibliography system which provides open bibliographic information on major computer science journals and proceedings. The data size of the latest XML data provided by the system in the form of single file reaches 3.5 GB. The famous Wikimedia [4] provides researchers with XML data export function. The size of XML data exported from its latest web documents exceeds 50 GB. OpenStreetMap [5] provides map data for thousands of websites, mobile applications and hardware devices. The exported data is stored in a single XML file, and the size of data exceeds 400 GB. The parsing and querying of XML data usually consumes a lot of resources, so large XML data volumes are difficult to handle in standalone systems, and multi-computer distributed processing becomes an inevitable requirement [6].

In XML data processing applications, on the one hand, XML query is the main function of data processing [7], and XPath [8] evaluation is the core part of XML query. On the other hand, XML documents have strict nested formatting constraints and need to be parsed before querying. Compared with the traditional relational structured data, the semi-structured characteristics of XML data make the parsing and query operations more complex. The

✉ Guorong Cai
guorongcai.jmu@gmail.com

1 Computer Engineering College, Jimei University, Xiamen, China

2 Digital Fujian Big Data Modeling and Intelligent Computing Institute, Xiamen, China

processing of large-scale XML data in distributed computing environment faces great challenges in feasibility and performance optimization. MapReduce programming model [9, 10] is widely used in large data processing to adapt to multi-machine distributed computing, and has achieved great success in general batch applications. However, the running environment configuration of MapReduce is cumbersome, and MapReduce can not directly and effectively deal with nested complex data such as XML. Due to a large number of iterative operations in XML processing, it is difficult to ensure good performance by multiple MapReduce operations [11]. XML data is semi-structured data organized in a nested way, it is difficult to be directly partitioned and processed. However, partition is often a prerequisite for adapting to distributed computing. Currently, most of the data partitioning in distributed XML processing is done in a pre-processing way. Typical examples, such as Fan et al. [12], partition XML data before processing distributed XPath queries and use serial XML parsing for data preparation. The preprocessing can not make full use of the advantages of distributed parallel computing on the one hand, but also reduces the overall automation. How to effectively integrate distributed processing of main stages such as XML parsing, indexing, and query has a positive impact on the overall performance of XML processing. From the specific parallelization techniques at each stage of XML processing, the XML parsing technology that supports arbitrary fragmentation [13] can effectively increase the flexibility of parallelization. Specific XML indexes [14, 15] play a positive role in improving the performance of XPath evaluation. In terms of query, XPath queries include navigational approach [16] and twig approach [17, 18], depending on how the query step is handled. Since the navigational XPath evaluation is easy to realize rich XML query semantics, it is of great significance to improve its efficiency through parallel processing [19]. Since XML processing involves key aspects such as XML parsing, indexing and querying, how to effectively integrate these aspects to support efficient distributed processing of XML data has become an important topic.

In response to the performance optimization problem and the difficulty of configuration during the processing of large-scale XML data in distributed computing environments, this paper develops a distributed processing technique for large XML data called the dXML method. Since dXML does not rely on complex environment configuration, the working mechanism of the method has the flexibility to support efficient regular and ad hoc queries [20] for XML large data in distributed environments. Our main contributions include:

(1) We give a way to support the distributed parsing and index creation of XML arbitrary fragments, and use relation index for efficient navigational XPath evaluation.
(2) We propose a filter-upon-pre-evaluate method for distributed XPath evaluation, which can fully localize the data during distributed XPath evaluation and reduce the communication overhead.
(3) We provide a one-stop solution that integrates distributed XML parsing, distributed index creation, and distributed XPath query to support automatic distributed processing of large XML data.
(4) We present the overall performance evaluation results and the comparative results of queries, showing the advantages of dXML method.

The rest of the content is organized as follows. Section 2 introduces the related work. Section 3 details the dXML method proposed in this paper, including its overall framework, distributed XML parsing, distributed index building, and distributed XPath query. Section 4 conducts performance evaluation and comparative experiments. Section 5 discusses the problems that exist. The last section provides conclusion and future work.

2 Related work

Since XML data models are relatively complex and semi-structured, and XML queries have rich semantics, efficient processing of XML queries is a challenge in the context of large-scale data. From the development of XML distributed processing system, MapReduce programming model, as a general parallel framework, has been widely used in the processing of distributed large data sets in recent years. The main distributed platforms, including Hadoop [21], Spark [22], are based on the MapReduce model. For example, HadoopXML [23] implements the simultaneous processing of multiple XML queries on Hadoop platform. Large XML data in this system is pre-partitioned into XML data blocks. Spark-XML [24] is an XML processing module integrated on Spark platform to support distributed XML query by converting XML data into DataFrames. Andromeda [25, 26] queries and updates large XML datasets through MapReduce cluster computing. The system partitions large XML data statically and dynamically, and XML partitions stored in the form of subtree set are distributed across workers to support distributed computing in clusters. However, the system does not support common XPath predicate evaluation. PAXQuery [27] implements parallel processing of XQuery [28] queries in clusters through a Parallelization Contract (PACT) programming model, but manual data partitioning is required for large XML datasets. VXQuery [29] is an open source large data processing platform for XML. It uses Hyracks data parallel processing platform and also does not support automatic partitioning of large XML data. OHX (Oracle XQuery for Hadoop) [30] converts XQuery queries into MapReduce work tasks on Hadoop to run on clusters, but the logic for migrating and transforming data depends on the developer's design. SparkSQL [31] uses the Spark platform for distributed query of XML data. The system requires expensive data conversion because it uses relational storage to store XML data. Moreover, common predicate evaluation is not supported at this time. From the above XML distributed processing systems, the existing systems support distributed processing to some extent, but there are still some limitations, such as the level of support for XML query semantics, the automated integration of various stages in the query process, and the distributed parsing ability of large XML data.

From the different stages of XML distributed processing, each research has its own focus since XML processing involves XML parsing, XML index and XML query. Khatchadourian et al. [32] developed a language called ChuQL, which can use MapReduce framework for XML distributed processing by extending XQuery. Fegaras et al. [33] designed MRQL language for the analysis and processing of large-scale XML data in MapReduce environment, and optimized the processing of XML queries through this language. Senk et al. [34] proposed an XPath distributed evaluation method, which is limited to simple paths and does not support common predicate evaluation. The HoX-MaRe method proposed by Damigos et al. [35] is a distributed XPath evaluation method. This method partitions XML data to horizontal fragmentations, combines query decomposition, and carries out distributed query in the way of MapReduce. The XML data in this method is pre-partitioned, and parallel XML parsing is not involved. Kunfang et al. [36] gave a keyword query method for large-scale XML data, which involves parallel loading of XML data. However, when partitioning and loading XML data, some partition parameters need to be provided in advance. Liang et al. [37] proposed a method based on NoSQL platform for the processing of massive small XML data. The method utilizes HBase [38] distributed platform through XML encoding conversion and query optimization. Similar to the work in [37], Liu et al. [39], when processing large XML-based biological datasets, also transfers XML data into HBase and requires transforming the XML query model into a MapReduce query model. These non-native-XML working models bring the overhead of data transformation. Longjian et al. [40] coordinates XML processing through shortest path routing algorithm to improve processing efficiency across the distributed network, but does not involve XML partitioning and generic XML queries. Bi et al. [41] presented a distributed twig query method for XML big data, which can support arbitrary partition, while the partition and index establishment are processed serially. Subramaniam et al. [42] proposed a distributed processing method of twig query, which requires path index of data. The query is decomposed by linear path, and then the local results are obtained by distributed evaluation. Finally, the merging process is optimized by pruning. Since the method is a non holistic twig query, it has a high merging cost. Fan et al. proposed TwigStack-MR method in [43], which performs distributed processing based on MapReduce framework for TwigStack [17], a classic holistic twig query method. In their latest work [12], they further explained and evaluated the method. However, their method still lacks the ability to parse XML fragments in parallel. The partition of XML data is an important prerequisite for distributed XML processing [44], which involves the parsing and processing of XML fragments. From the perspective of the overall process, the optimization of XML parsing, which is usually ignored, is an important factor causing performance bottleneck. Choi et al. [45] explored the distributed parallel parsing of large-scale XML data and provided a distributed approach to obtain common XML encodings, including interval-based encoding [46] and prefix-Based encoding [47]. However, their work does not involve XML queries. Hsu et al. [48] proposed a method of creating an index for large XML documents in MapReduce using cloud parallel computing, and then to perform XPath queries on this basis. In the process of XML parsing, each part file of the document is processed by MapReduce in sequence. Such serial processing between part files is likely to cause performance bottlenecks. In addition, the work did not test the frequently used XPath predicate query, so the feasibility of predicate branch query is unknown. From a variety of related work, more practical integrated distributed solutions involving XML parsing, XML indexing and XML query in XML processing still need to be developed. Since the optimal design of distributed XML processing involves several related topics, a list of the main related works by research topic is given in Table 1 in order to facilitate the clarification of the entire technical system.

Table 1 Taxonomy of related works in research topics

Topic	Authors	Description
XML system	Choi et al. [23]	HadoopXML system, based on the MapReduce framework on Hadoop Spark-XML system, based on the MapReduce framework on Spark
	Owen et al. [24]	
	Bidoit et al. [25, 26]	Andromeda system, based on the MapReduce framework
	Camacho-R et al. [27]	PAXQuery system, based on the PACT programming model
		VXQuery system, based on Hyracks data parallelism platform
	Carman et al. [29]	OHX system, based on the MapReduce framework
	Oracle Inc. [30]	SparkSQL system, based on the MapReduce framework
	Hricov et al. [31]	
XML parsing	Chen et al. [13]	Support parallel parsing of arbitrary XML fragments
	Braganholo et al. [44]	Overview of distributed XML fragmentation
		Distributed parsing of XML data
	Choi et al. [45]	
XML indexing	Chen et al. [14, 15]	Parallel construction of relation index
	Lu et al. [47]	Indexing for extended Dewey prefix encoding
	Hsu et al. [48]	Indexing for interval encoding under distributed conditions
XML query	Khatchadourian et al. [32]	ChuQL language, extending XQuery, running on MapReduce
		MRQL language to optimize distributed queries
	Fegaras et al. [33]	HoX-MaRe method, the typical distributed navigational method
	Damigos et al. [35]	TwigStack-MR method, the typical distributed Twig method
	Fan et al. [12, 43]	

3 Proposed method

3.1 Overview of dXML method

The dXML method proposed in this paper is an integrated solution for distributed parsing and query of a single large XML document. The typical running environment of dXML is shown in Fig. 1. The cluster in the dotted box contains two types of machines with different roles: coordinator and worker. There is only one coordinator, which plays the role of coordinating the workers in the cluster and undertakes global computing tasks. There are multiple workers, which allow heterogeneous configuration and can be expanded in quantity. Their role is to perform distributed computing tasks. Clients outside the cluster interact with the coordinator through LAN or Internet, submit query requests and required data to the coordinator, and obtain query results from the coordinator.

The distributed processing procedure of dXML is shown in Fig. 2. The whole procedure includes two stages: the first stage is distributed data preparation, and the second stage is distributed query. The first stage mainly involves XML parsing and index creation. The coordinator loads the XML document specified by the client, partitions the document according to the number of workers, obtains the XML fragment (see Definition 1), and then directly sends the fragment to each worker, so that each worker obtains a

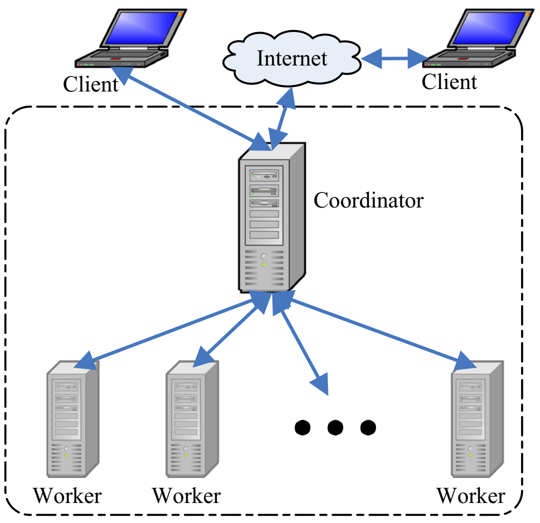
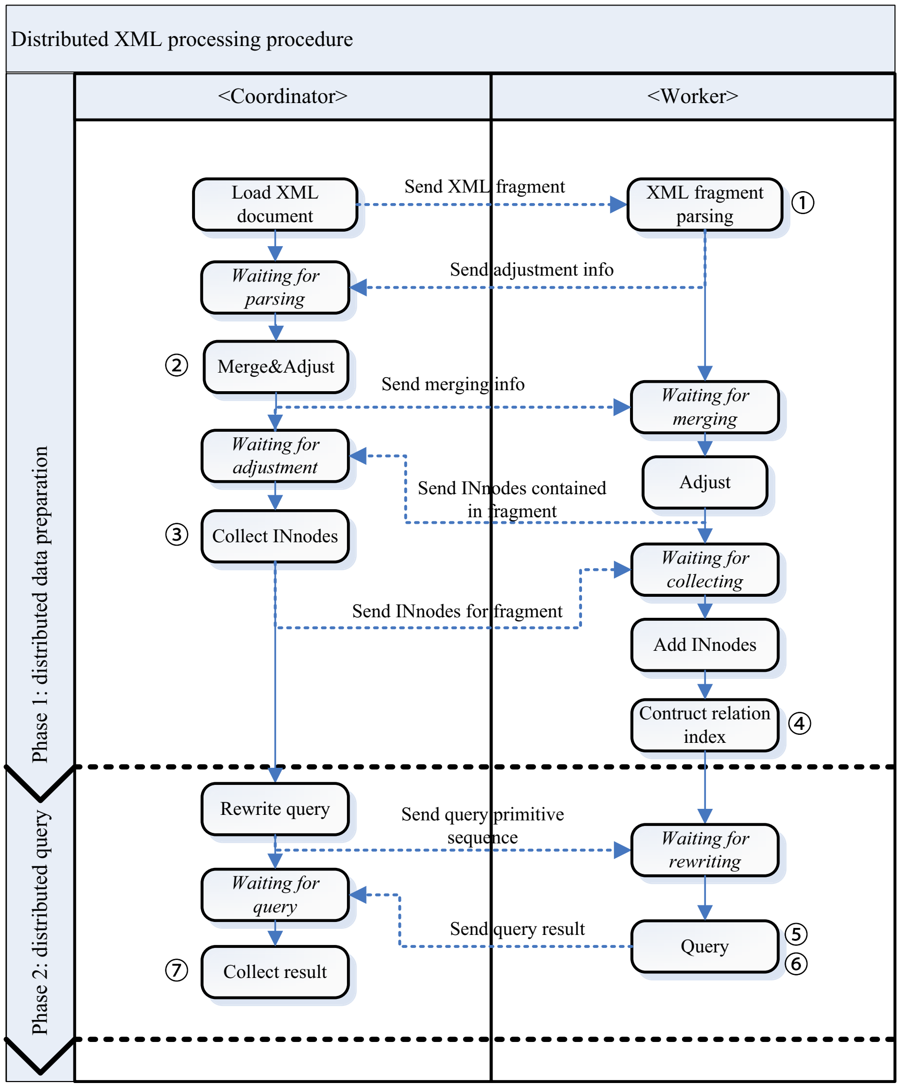


Fig. 1 Running environment of dXML

fragment. After receiving the XML fragment, the worker starts the local XML parsing, which is usually the most time-consuming. However, due to the distributed parallel processing of each worker, it can effectively overcome the problem of poor performance caused by serial XML parsing. When each worker completes the parsing of XML fragment, it sends the initial adjustment information back to the coordinator. After collecting all the adjustment information, the coordinator starts merge operation to



① Refer to Algorithm 2.
② Refer to Algorithm 3.
③ Refer to Algorithm 4.
④ Refer to Algorithm 5.
⑤ Refer to Algorithm 6.
⑥ Refer to Algorithm 7(a).
⑦ Refer to Algorithm 7(b).

Fig. 2 Distributed processing in dXML

obtain the merging information that can be used for further adjustments. Then the merging information is sent to each worker to guide the adjustment of the local parsing result. Since the subtrees obtained from the parsing result of XML fragment is local, the INnode (see Definition 4) information is required to construct a fragment tree (see Definition 5). The detailed information of the INnodes exists in the precursor fragment. Therefore, the INnode information owned by each fragment is collected by the coordinator, so as to further be sent to the corresponding worker through the coordinator. After receiving the detailed information of the required INnodes, the worker completes the construction of fragment tree. The worker creates the corresponding relation index on the basis of completing the construction of fragment tree. So far, the distributed data preparation has been completed. In the second stage, the coordinator first preprocesses the query submitted by the client. For the XPath query expression, the query primitive sequence is generated by rewriting, and then the sequence is sent to each worker for query processing. While for XQuery programs, preprocessing steps such as XPath expression extraction are also required. Next, the filter-upon-pre-evaluate approach proposed by us is used for distributed query. After each worker completes the local query, it sends the results back to the coordinator, and the coordinator collects all the local results and returns the final query results to the client.

The corresponding processing programs are deployed on the coordinator and the workers respectively to complete the distributed processing in a synchronous manner. From the perspective of the overall interaction framework, the main process on coordinator is described in Algorithm 1(a), and the main process on worker is described in Algorithm 1(b).

3.2 Distributed XML parsing

The steps of distributed XML parsing in dXML method include XML document partition on the coordinator, fragment parsing on the worker, and merging and adjustment on the coordinator. Finally, the construction of fragment tree is completed on each worker. Because our method supports the parsing of arbitrary XML fragments, the overall performance is improved through the distributed parallel parsing of fragments.

Algorithm 1 Interactive framework for distributed processing

```
Algorithm 1 (a) On coordinator:
Input: XML document data $D$ and query $Q$
Output: XML query result $S$
 1: Sync { // Start synchronization.
 2:    SendFragment($D$);    //Send XML fragment to each worker.
 3:    wait;    // Waiting for parsing results.
 4:    MakeMerge( );    // Merge XML subtrees.
 5:    wait;    // Waiting for adjustment results.
 6:    CollectINnodeDetail( );    // Collect INnode information.
 7:    SendQuery($Q$);    // Send query request.
 8:    wait;    // Waiting for local query results.
 9:    $S$←CollectResult( );    // Collect local query results.
10: } // End synchronization.
11: return $S$;
Algorithm 1(b) On worker:
 1: Sync { // Start synchronization.
 2:    wait;    // Waiting for XML fragment.
 3:    ParseFragment ( );    // Parse XML fragment.
 4:    wait;    // Waiting for merge results.
 5:    AdjustFragment ( );    // Adjust XML subtrees.
 6:    wait;    // Waiting for INnode information.
 7:    AddINnode( );    // Add INnodes and construct fragment tree.
 8:    CreateIndex( );    // Create relation index.
 9:    wait;    // Waiting for query request.
10:    DoQuery( );    // Execute query.
11: } // End synchronization.
```

Considering that the dXML method includes three main steps: distributed XML parsing, distributed index construction and distributed XPath query, they are described in detail below.

3.2.1 Arbitrary XML fragment

Large XML data needs to be distributed across workers for parsing. A simple method of partitioning is to divide the entire XML data equally by the number of workers, so that a worker can process one fragment. This partitioning, regardless of the logical structure of XML, is arbitrary based only on the absolute position in the XML document. Such partitioning can be accomplished efficiently and can also achieve a certain load balancing.

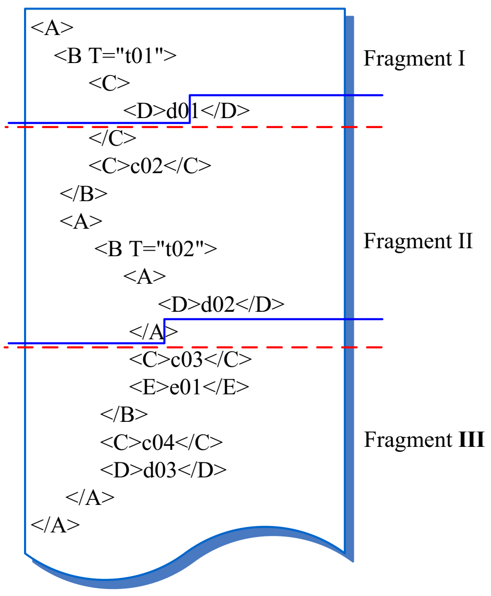
Definition 1 (*XML fragment*) Refers to continuous XML data within a certain range of an XML document. It is a logical partition on XML document data, which is still unparsed text.

Definition 2 (*Node*) Refers to the basic components that make up the XML document tree. It mainly includes element nodes and attribute nodes. The node storage in this paper adopts interval-based encoding [42], which is described as $< ID$, *nodeType*, *tagName*, *level*, *startpos*, *endpos* $>$, including six items: *ID*—node ID, a unique sequence number in document order; *nodeType*—node type, *nodeType* $\in$ {ELEM, ATTRIB}, corresponding to the two main types of elements and attributes; *tagName*—the tag name of node; *level*—the level of the node in the DOM tree; *startpos*—the document position where the node starts and *endpos*—the document position where the node ends. The "document position" here refers to the file pointer position expressed by byte offset. The content information of the node is recorded separately using a hash table.

For example, in Fig. 3b, the encodings of nodes B1, C1 and T2 are $<$ 1, ELEM, B, 0, 5, 57 $>$, $<$ 3, ELEM, C, 2, 18, 39 $>$ and $<$ 8, ATTRIB, T, 3, 67, 74 $>$ respectively.

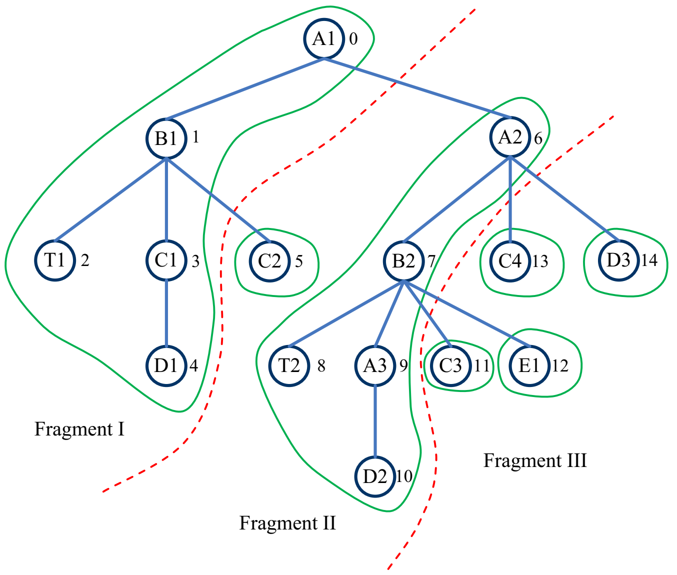
Definition 3 (*Subtree*) Refers to part of the XML document tree. The subtree types obtained from fragment parsing include complete subtree and incomplete subtree. Complete subtree means that all information of the subtree is in the same fragment. The incomplete subtree is truncated, and some node information is in different fragments. The storage structure of subtree is described as $< ID$, *nodes* $>$, including two items: *ID*—subtree ID, a unique sequence number and *nodes*—list of nodes contained in subtree.

Arbitrary partitioning of an XML document using document position rather than physical partitioning. The method is to set the file reading pointer directly to the starting position of each fragment and start to read, and then adjust the fragment boundary as necessary to obtain the fragment boundary information. This is a lightweight partitioning method since complex preprocessing is not necessary. For example, the XML document in Fig. 3a is arbitrarily partitioned into three fragments, and the blue solid line represents the document position of fragment. After boundary adjustment, the actual partition will be carried out according to the principle of retaining complete tag names, and the red dotted line indicates the adjusted boundary. Figure 3b is the corresponding XML document tree. Each circle represents a node, and the string in the circle represents the tag name of the node, which is distinguished by numbers. The number next to the circle represents the node ID which reflects the document order. The red dotted line indicates the boundary of fragment. The


Fragment I
Fragment II
Fragment III

```
<A>
  <B T="t01">
    <C>
      <D>d01</D>
    </C>
    <C>c02</C>
  </B>
  <A>
    <B T="t02">
      <A>
        <D>d02</D>
      </A>
      <C>c03</C>
      <E>e01</E>
    </B>
    <C>c04</C>
    <D>d03</D>
  </A>
</A>
```

(a) XML document and fragments



(b) XML document tree

Fig. 3 Arbitrary partitioning of XML

nodes within the green solid line belong to the same subtree, and each fragment may contain multiple subtrees.

3.2.2 Parallel parsing of XML fragments

Parsing an XML fragment is a process of constructing subtrees in parallel on each worker in a distributed manner. Subtree construction not only obtains the subtree set of the fragment, but also obtains the mismatch head/tail tag sequence of each fragment. The subtree information retrieved within the fragment, including the node relation in the local subtree, the node content, and the tag name. In the process of subtree construction, the node to be processed at the beginning of each fragment is the root node of the first subtree, and the root node of the subtree can only be element type. By recording the necessary mismatch information, the dependence between each fragment is decoupled, so that each fragment can be parsed independently and processed in parallel by different worker. Each worker maintains a tag stack within the fragment for mismatch tag identification and subtree root node identification.

Algorithm 2 gives a brief description of the subtree construction process. The overall framework of this process is to continuously read a complete tag for a fragment until the end, during which all the subtrees within an XML fragment are constructed by parsing the content of the tag.

constructed by parsing the content of the tag.

```
Algorithm 2 Performing subtree construction on the worker
Subtree []    ConstructSubtrees (Fragment fm)
Input: fm - fragment information
Output: subtree set
1: subtreeSet←Φ; // Local variable: Subtree [] subtreeSet
2: pos←fm.startPos;
3: while (pos≠fm.endPos) do
4:     T<pos, completedTagString>←ReadCompletedTag(fm, pos, fm.endPos);
5:     P<stack, subtree>←ParseTagContent(T.completedTagString);
6:     subtreeSet←subtreeSet ∪ P.subtree;
7:     pos←T.pos;
8: end while
9: subtreeSet←ProcessUnmatched(P.stack);
10: return subtreeSet;
```

The line 3～8 in Algorithm 2 describes that in the fragment range, first use the *ReadCompletedTag* function to get a complete tag string, then call the *ParseTagContent* function to parse the tag string to get the information of the subtree, and use the stack to record the mismatch tag information, repeat such operations until the end of the fragment. *T* in line 4 is a local variable in tuple form, which is used to record the complete tag string and the current read position. *P* in line 5 is also a local variable in tuple form, which is used to record tag stack and subtree information. Line 9 calls the *ProcessUnmatched* function to process the mismatch tag information. Since the last subtree may be incompleteee, it needs to be built here and the relevant information of the subtree set needs to be updated. The algorithms involved in each processing function are omitted here.

3.2.3 Merging and adjustment

Since the parsing results of each fragment are independent of each other, some information is local and inconsistent with the global XML tree, such as the ID value, the level value, the actual start and end position of the node, etc. the real value of these information needs to be obtained through merging and adjustment. The adjustment should be carried out not only on the coordinator, but also on each worker. First, merge and adjust the global relevant information on the coordinator, and then send the merging results to the workers to guide the local adjustment. Finally, each worker completes local adjustment according to the merging results.

The merging and adjustment in the coordinator includes several steps: first, initialize the input and output information, collect the adjustment information from each working machine and create the tail tag index; secondly, by finding the parent nodes of the subtrees and adjusting the relevant information, the preliminary final subtrees are obtained; next, merge the tag name table; finally, pack the merging information. The adjustment information collected by the coordinator from each worker includes the level values of nodes, the mismatch head and tail tag information, the number of subtrees and the number of nodes. While the merging information returned to each worker after merging includes the adjusted document tail position shared by each fragment and the level values of all nodes. Algorithm 3 briefly describes the merging and adjustment process.

```
Algorithm 3 Perform merging and adjustment on the coordinator
MergeInfo    MergeAndAdjust (AdjustmentInfo aj)
Input: aj - adjustment information from the workers
Output: merging information
1: mg←Φ; // MergeInfo mg.
2: Initial(aj, mg); // Initialize input and output information.
3: BuildTailTagIndex(aj); // Create the tail tag index.
4: FindParentOfSubtreeAndAdjust( ); // Find the parent nodes of the subtrees and adjust the relevant information.
5: MergeNameTable( ); // Merge tag name table.
6: mg←PackMergeInfo( ); //Pack the merging information.
7: return mg;
```

In Algorithm 3, the *FindParentOfSubtreeAndAdjust* function is called to search the parent node of the subtree. The basic principle is: in the node sequence of mismatch head tag, find the node with the mismatch head tag closest to the root node of the subtree in document order, that is, the parent node of the subtree. The principle is illustrated in Fig. 4. The first row of blocks in the figure represents the mismatch tag sequence obtained from the parsing of the fragments in Fig. 3. The string in the box represents the node where the mismatch tag is located, and the ’ < ’ symbol indicates the mismatch head tag, and the ’ > ’ symbol indicates the mismatch tail tag. The dashed vertical line is the partition position. For the subtree composed of node C2, the root node is C2, and the position of its head tag is shown by the first arrow. Its closest unpaired mismatch head tag is ’B1 < ’, so the parent node of the subtree is node B1. For the subtree composed of nodes A2, B2, T2, A3 and D2, the root node is A2, and its head tag position is shown by the second arrow. Since there are paired tags ’B1 < ’ and ’B1 > ’ in the first half of the interval, the closest unpaired mismatch head tag should be ’A1 < ’, so the parent node of the subtree is A1. Similarly, the parent node of node D3 is A2.

3.2.4 Construct fragment tree

According to the merging information sent by the coordinator, each worker completes the adjustment of the local parsing results and obtains the adjusted subtree set. In order to facilitate query, each worker needs to build a complete local tree based on these subtree sets. Related definitions are introduced below.

Definition 4 *(Incomplete node, abbreviated as INnode)* Refers to those nodes whose node information is distributed to different fragments when the XML document is partitioned. INnode is not obtained from the parsing of the current fragment, but plays the role of logically connecting to the root node of the original document tree.

INnodes are related to specific fragments. If the XML document is partitioned according to Fig. 3, nodes A1, B1,

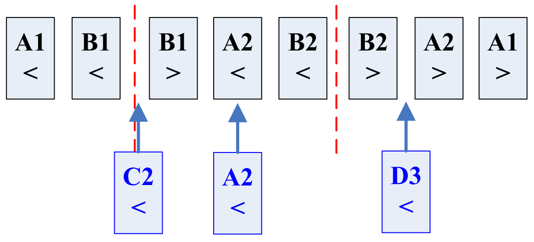
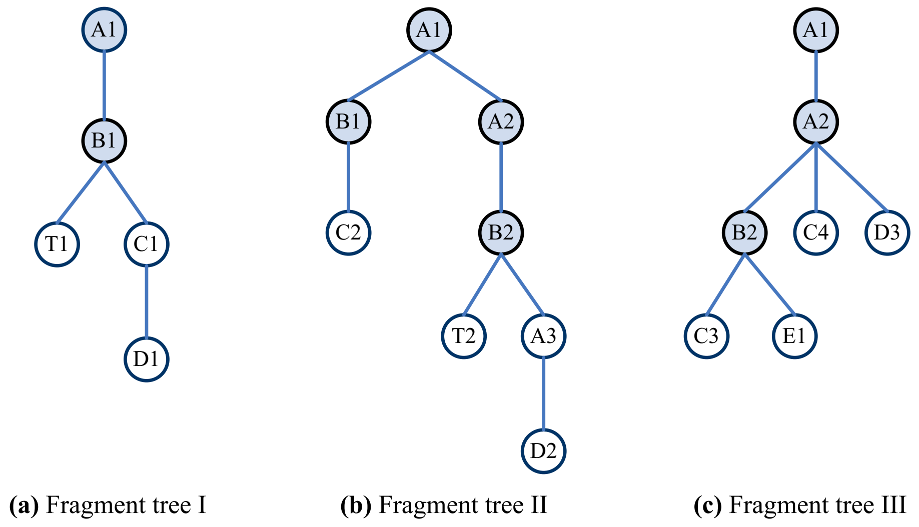


Fig. 4 Example of finding the parent node of subtree

Fig. 5 Fragment tree



A2 and B2 in Fig. 3b are INnodes. For ease of description, "complete node" (abbreviated as CNnode) is used to indicate that the node is not partitioned, but completely exists in a fragment. For example, nodes C1, C4 and D3.

Definition 5 (*Fragment tree*): Refers to a complete local document tree formed after adding INnodes information, which takes the root node of the original document tree as the root and contains all nodes in the fragment.

Figure 5 shows the corresponding fragment trees for each fragment in Fig. 3, with the INnode with gray background color.

Proposition 1 *All ancestors of the first node in a non-first fragment are the additional INnodes required to construct the fragment tree for that fragment.*

Proof If node $u1$ is the first node in the fragment, since $u1$ must be the root node of the first subtree $T1$ of the fragment, all ancestors of $u1$ form a INnode sequence, which is recorded as P_{u1}. Assuming that there is a root node $u2$ of another subtree in the fragment, and one of its ancestor nodes $u3$ satisfies $u3 \notin P_{u1}$, there is a path from $u1$ to the root node $u0$ of the original document tree and a path from $u2$ through $u3$ to $u0$. Since $u1 < u2$ (indicating $u1$ is before $u2$ in document order), $u1 < u3$ in the document tree. On the other hand, because $u1$ and $u2$ are in the same fragment, $u3$ is the ancestor of $u2$ and must be in the precursor fragment, so $u3 < u1$. This conclusion contradicts the previous conclusion, so the previous assumption does not hold. It shows that the ancestor nodes of the root nodes of other subtrees except $T1$ do not add additional INnodes, that is, all ancestors of node $u1$ are all the additional INnodes required. □

As can be seen from Fig. 3b, for fragment II, the first node is C2, and all its ancestors are B1 and A1. For fragment III, the first node is C3, and all its ancestors are B2, A2 and A1. A key to the construction of the fragment tree is to obtain the required INnodes. After the merging and adjustment are completed on the coordinator, a list containing all the adjusted nodes is obtained. Since the adjusted node ID and level values are globally unified, the INnodes of all fragments can be found on this basis. All ancestors of a node $u1$ can be described in the form of a recursive predecessor parent node as $P_{u1} = \{u \mid u \leftarrow u1, u \leftarrow Parent(u)\ \textbf{till}\ u = u0\}$, which calls the *Parent* function to get the parent node, and $u0$ refers to the root node of the original document tree. According to Proposition 1, the method to obtain the required INnodes is as follows: for a given fragment, take the first node of the fragment as the seed node, traverse backward according to the document order, and obtain the precursor parent nodes one by one until the root node of the original document tree, which is the INnodes of the fragment. As described in Algorithm 4.

```
Algorithm 4 Perform the INnodes search on the coordinator
Node[ ][ ]   FindINnode ( )
Input: (Expressed in local form) the first node of each fragment Node u1 and each adjusted XML node Node u
Output: Node [][] innode - INnodes sequence. Note: innode[i] represents all INnodes of fragment i
1: INnode←Φ;
2: foreach i∈{1 to N-1}   //N is the number of fragments; i=1 starting from the second fragment.
3:    u1←GetFirstNode(i);   //Get the first node of the fragment.
4:    foreach u∈{u1 to u0}   // u0 is the root node of the original document tree, defined as Node u0.
5:       if (u.level = u1.level-1)   // Condition: u is the parent node of u1.
6:          INnode[i]←INnode[i]∪{u};
7:          u1←u;
8: return INnode;
```

Lines 2 to 7 in Algorithm 4 perform the acquisition of INnodes from the second fragment to the last fragment one by one. It is noted that the first fragment does not need to obtain INnodes. Line 4 indicates traversal from the current node to the root node in reverse order of the document order. After finding and collecting all the INnodes on the coordinator, these INnodes will be sent to the worker in need. After receiving these INnodes, the worker updates the local nodes list in order by header insertion, that is, INnodes are inserted before the local nodes according to the document order. In this way, the construction of a fragment tree is completed on each worker.

3.3 Distributed index construction

After the XML parsing is completed, the relation index of XML nodes needs to be further constructed, which provides a way to optimize query processing for the navigational XPath evaluation method based on relation search.

Definition 6 (*Relation index*) Refers to the storage structure that records the effective relation between XML nodes. The entry of index is represented by a tuple as $< u.ID, v.ID, r_{u\to v} >$, where $r_{\mathrm{u}\to\mathrm{v}}$ represents the unique relation type value of node *u* and node *v*, and $r_{\mathrm{u}\to\mathrm{v}} \in$ {DE, CH, AT}, which contains the three common node relation types: descendant, child and attribute. The relation index of a node *u* refers to the set of index entries of the node and all subsequent nodes in document order that have a relation with the node, expressed as *RIndex*: $\bigcup_{u.\mathrm{ID} < vi.\mathrm{ID}} \{< u.ID, vi.\mathrm{ID}, r_{u\to vi} >\}$.

For example, in Fig. 3b, the index of node B1 is {< 1, 2, AT > , < 1, 3, CH > , < 1, 4, DE > , < 1, 5, CH >}, the index of C1 is {< 3, 4, CH >} and the index of T2 is {}. Since the fragment tree is a complete tree with the root of the original document tree as its root, the relation index can be created directly on the fragment tree. Suppose *u* and *v* are XML nodes with interval-based encoding, and the basic rules for relation calculation are as follows:

① $r_{u\to v}$ = 'CH' **if** (u.begin < v.begin)∧(v.begin < u.end)∧(u.level = v.level-1)∧(v.nodeType = ELEM).
② $r_{u\to v}$ = 'DS' **if** (u.begin < v.begin)∧(v.begin < u.end)∧(u.level ≠ v.level-1)∧(v.nodeType = ELEM).
③ $r_{u\to v}$ = 'AT' **if** (u.begin < v.begin)∧(v.begin < u.end)∧(u.level = v.level-1)∧(v.nodeType = ATTRIB).

In rule ②, since only one relation is stored between two nodes, and 'DS' contains 'CH' semantically, the constraint of "u.level ≠ v.level-1" is considered when creating the index. During implementation, 'CH' relation information will be automatically included for 'DS' relation. In order to optimize processing, an auxiliary relation 'NN' is added to indicate that there is no relation between nodes. The construction of relation index is a process of calculating the relation value. This process uses the relation calculation rules to get the relation value between two nodes, and then stores the relation value in the relation index. The index creation executed on each worker is shown in Algorithm 5.

In line 5 of Algorithm 5, the *GetRelation* function is called to calculate the relation value according to the

```
Algorithm 5 Perform relation index creation on the worker
RIndex    CreateRIndex (Node[ ] NC)
Input: NC – the sequence of XML nodes in the fragment
Output: RIndex index - relation index
1: index←Φ;
2: N←NC.length; //N is the number of nodes.
3: foreach node id i∈{I| 0<I<N-1}
4:     foreach node id j∈{J|i+1≤J≤N-1}
5:         r←GetRelation(NC[i] , NC[j]);
6:         if(r=DS ∨ r=CH ∨ r=AT)
7:             index←index∪{<i, j, r>};
8:         if(r=NN) break;
9: return index;
```

relation calculation rules. When the 'NN' value is obtained, the subsequent relation value must be 'NN' [17]. The optimization processing in line 8 can greatly reduce the unnecessary traversal and improve the performance of index creation. Since each worker independently creates the corresponding relation index according to the parsing and adjustment results of local fragment, the index creation is a process of distributed parallel processing as a whole.

3.4 Distributed XPath query

In the navigational query based on relation index [13, 17], the query steps of XPath are organized by the sequence of query primitives, which is efficient and flexible, and easy to implement the query semantics of XPath. However, in the distributed environment, the locality of XML data may lead to evaluation errors. For example, when evaluating XPath predicates, some conditions may not be satisfied because the data is distributed to different workers, resulting in the loss of evaluation results. In order to adapt to the distributed computing environment, a distributed XPath evaluation approach based on relation index is proposed below. It currently covers the functions of XPath subsets {/, //, *, @, []} and supports the operation of complex nested multiple predicates.

3.4.1 XPath query primitive

In the distributed query process of each worker, XPath evaluation based on relation index is used to organize the query in the form of query primitive sequence. Query primitives are the implementation of corresponding XPath query steps. The supported query primitives include non-filter primitives and filter primitives. Non-filter primitives correspond to the axis operation of XPath, including primitives for descendants, children and attributes. Filter primitives correspond to predicates of XPath, including the basic filter primitive *FilterS1ByS2* and variants of filter primitives, such as filter primitives with 'AND', 'OR' and 'NOT' conditions. The evaluation of query primitives is described by Algorithm 6. In line 5 in Algorithm 6(a) and line 3 in Algorithm 6(b), *n* is an index of node *m*, and *index* is the relation index, which is defined as *RIndex index*. Algorithm 6(a) is an evaluation description of a non-filter primitive case, taking the primitive of getting descendant nodes as an example. Algorithm 6(b) describes the evaluation of a basic filter primitive. Its working mechanism is to filter the node sequence *input1* according to the node sequence *input2*. XPath is evaluated through index lookup and simple comparison, so it has high efficiency.

Algorithm 6 Perform query primitive evaluation on the worker

Algorithm 6(a) Non-filter primitive case:

```
Node[ ] GetDescendant (Node[ ] input, String tagName)
Input: input - XML node sequence; tag Name - tagName to be checked in XPath
Output: Node [] result - XML node sequence
1: result←Φ;
2: if (tagName ="*") nameTest←false;    //Whether to perform node detection.
3: else nameTest←true;
4: foreach m∈ input    //m is an input node.
5:     foreach n∈ index(m.ID)
6:         if ((nameTest = true ∨ (NC[n.vi.ID].tagName = tagName))∧ (n.r =DE ∨ n.r =CH))
7:             result←result∪ {NC[n.vi.ID]};    // NC is the sequence of XML nodes in the fragment, defined as Node [] NC.
8: return result;
```

Algorithm 6(b) Filter primitive case:

```
Node[ ] FilterS1ByS2 (Node[ ] input1, Node[ ] input2)
Input: input1 and input2 - XML node sequence
Output: Node [] result - XML node sequence
1: result←Φ;
2: foreach m∈ input1;    //m is an input node.
3:     foreach n∈ index(m.ID)
4:         if (n.vi.ID∈ {input2.ID})
5:             result←result∪ {m};
6:             break;
7: return result;
```

The corresponding query primitive sequence is obtained by rewriting the input XPath expression on the coordinator, and then the sequence is sent to each worker for evaluation. Describe the rewriting function with $\boldsymbol{T}[e]_s = e'$, where e is the XPath expression, e' is the rewritten primitive sequence, and s represents the input XML node sequence in the current context. The main rules of XPath query rewriting are as follows, where e_{head} represents the header of expression e, corresponding to a tag name, and e_{tail} represents the rest after e_{head} is removed.

① $\boldsymbol{T}[//e]_{s0} = \boldsymbol{T}[e_{tail}]_{s1}$ **where** {s1←GetDescendant (s0, e_{head})}

② $\boldsymbol{T}[/e]_{s0} = \boldsymbol{T}[e_{tail}]_{s1}$ **where** {s1←GetChild (s0, e_{head})}

③ $\boldsymbol{T}[@e]_{s0} = \boldsymbol{T}[e_{tail}]_{s1}$ **where** {s1←GetAttribute(s0, e_{head})}

④ $\boldsymbol{T}[[e]]_{s0}$ = FilterS1ByS2 (s0, s1) **where** {s1←$\boldsymbol{T}[e]_{s0}$}

⑤ $\boldsymbol{T}[[e1 \text{ and } e2]]_{s0}$ = FilterS1ByS2_AND (s0, s1, s2) **where** {s1←$\boldsymbol{T}[e1]_{s0}$, s2←$\boldsymbol{T}[e2]_{s0}$}

⑥ $\boldsymbol{T}[[e1 \text{ or } e2]]_{s0}$ = FilterS1ByS2_OR (s0, s1, s2) **where** {s1←$\boldsymbol{T}[e1]_{s0}$, s2←$\boldsymbol{T}[e2]_{s0}$}

⑦ $\boldsymbol{T}[[not(e)]]_{s0}$ = FilterS1ByS2_NOT (s0, s1) **where** {s1←$\boldsymbol{T}[e]_{s0}$}

For example, if the input XML node sequence is *s0*, the query primitive sequence obtained from the rewriting result of query "/A[/B[//C]]//D" is organized as follows:

```
s6←GetDescendant (s5, D)
    where{ s5←FilterS1ByS2 (s1, s4)
        where { s4←FilterS1ByS2 (s2, s3)
            where { s3←GetDescendant (s2, C)
                where { s2←GetChild (s1, B)
                    where { s1←GetChild (s0, A)
            }}}}}
```

3.4.2 XPath query based on filter-upon-pre-evaluate

When the query primitive sequence and the data to be queried are available, each worker executes the query and then returns the query results to the coordinator for merging. When querying on each fragment tree, the results obtained may be duplicate, thus it is necessary to further de-duplicate during merging. On the other hand, predicate evaluation may have problems in distributed processing. Taking the fragment trees in Fig. 5 as an example, assuming that "//B[//C]//D" needs to be queried, the correct result should be {D1, D2}. However, if each fragment tree is queried independently, the query results obtained from the three fragment trees are {D1}, {} and {} respectively, and the merging final result is {D1}. On fragment II, node B2 is partitioned, and its child node C3 is distributed into fragment III, resulting in that the predicate condition cannot be met, so the returned result is null. In order to overcome this problem, we propose a distributed XPath evaluation approach based on filter-upon-pre-evaluate. The main idea is to pre evaluate each local fragment tree on the worker, record the corresponding predicate filter conditions, and then filter the pre evaluating results according to the merged filter conditions on the coordinator to obtain the final results that meet the conditions. Pre-evaluation and condition acquisition on each fragment satisfy data locality, thus effectively reducing network traffic. In the process of filter condition recording, value chain is used for data operation. For the convenience of explanation, the following definitions such as value chain are introduced.

Definition 7 (*INnode to be filtered, abbreviated as INnodeFt*) Refers to the input INnode in XPath predicate evaluation.

For example, when querying each XML fragment in Fig. 3, if the predicate query expression is "//B[]", the input nodes B1 and B2 are INnodes, so they are INnodeFts. For an XPath expression, according to the order of predicate, the input INnode corresponding to the first occurrence of predicate evaluation in nested predicates is called the first level INnodeFt, and so on. For example, in query "//A[/B[//C]]", the INnodes corresponding to the evaluation of "//A[]" are the first level INnodeFts, and the INnodes corresponding to "/B[]" are the second level INnodeFts or the last level INnodeFts in this case.

Definition 8 (*Branch path*) Refers to the linear query path formed from the root item to the leaf item in the query tree. In particular, the branch path containing the returned query item is called the main path.

For example, in the XPath expression "A[/B[//C]]//D", the branch paths include "A/B//C" and "A//D", where "A//D" is the main path. In the expression "A[/B]//C[D/E]/F", the branch paths include "A/B", "A//C/D/E" and "A//C/F", where "A//C/F" is the main path.

Definition 9 (*Value chain*) Refers to the value sequence used to record the evaluation results of branch paths. The items in the sequence are INnodeFts or the query results. The form of value chain is $< vn\text{-}1, vn\text{-}2, \ldots, v0 >$, and the corresponding node order from node *v0* to *vn-1* is increasing in the branch path. The last item *v0* in the sequence is the first level INnodeFt. The first item *vn-1* is the returned result item (for the main path) or the last level INnodeFt (for other branch paths).

The filter-upon-pre-evaluate approach is described as Algorithm 7, including the processing on the worker and coordinator. In line 1 of Algorithm 7(a), the fragment is evaluated normally according to the complete path of the query Q, and the CNnode evaluation result S_{CN} is obtained. Lines 2$\sim$4 evaluate the branch paths corresponding to query Q to obtain the pre evaluating result Sp. At the same time, record the filtering condition Sv required for the INnodeFts. In the process of branch path evaluation, the evaluating results are compared with the INnode list from parsing to obtain the INnodeFts. Line 5 sends the CNnode evaluating result, pre evaluating result and filter condition information to the coordinator. Line 2 in Algorithm 7(b) obtains the processing results from each worker. Lines 3$\sim$4 collect the filter conditions by branch path. Line 5 merges conditions in order. The order rule is to merge the predicates at the last level first, and then deal with each predicate one by one at the front level. Line 6 filters the node sequence of the pre evaluating result with the merging conditions to obtain the evaluation result of the INnodeFts. In line 7, the evaluation result of the INnodeFts and the evaluation result of the CNnodes are merged, and then de-duplication and sorting are carried out.

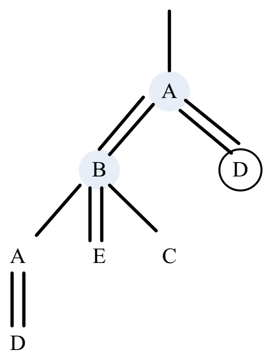


Fig. 6 Query tree

During the query process, since the query on each fragment is carried out according to the basic method of navigational evaluation, only the filter conditions are recorded during the query period, without data exchange between fragments. This process can be carried out independently in each worker, avoiding the communication between workers, thus ensuring the efficiency of predicate evaluation. During XPath evaluation, complete predicate filtering conditions ensure that the final result of predicate evaluation is not missed. Proposition 2 illustrates the availability of complete results from the evaluation of INnodeFts.

Proposition 2 The predicate filter condition in dXML is a sufficient condition to obtain the complete evaluation result of the INnodeFts.

```
Algorithm 7 Distributed XPath evaluation based on filter-upon-pre-evaluate
dXPath ($S_{input}$, $Q$)
Input: $S_{input}$ - XML node sequence; $Q$ - XPath query
Output: $S_{output}$ - XML node sequence
Algorithm 7 (a) Performs distributed processing on each worker:
1: $S_{CN}$←DoQuery($S_{input}$, $Q$);   //$S_{CN}$ is the CNnode evaluation result in the form of node sequence.
2: foreach $Qp \in Q$   //$Qp$ is a branch path.
3:     $Sp$←DoQuery($S_{input}$, $Qp$);   //$Sp$ is the pre evaluating result in the form of node sequence.
4:     $Sv$←GetFilter( );   //$Sv$ is the local filtering condition in the form of value chain set.
5: Send($S_{CN}$, $Sp$, $Sv$);

Algorithm 7 (b) Processing on the coordinator:
1: $S_{output}$←Φ;
2: $S_{CN}$[], $Sp$[], $Sv$[]←Receive();
3: foreach $Qp \in Q$   //$Qp$ is a branch path.
4:     $Svc$←CollectVC( );   //$Svc$ is the collecting result in the form of value chain set.
5: $Svm$←MergeVC($Svc$);   //$Svm$ is the merging result in the form of value chain set.
6: $S_{IN}$←SetFilter($Sp$[], $Svm$);   //$S_{IN}$ is the evaluation result of the INnodeFt.
7: $S_{output}$←PostProcess($S_{CN}$[], $S_{IN}$);
8: return $S_{output}$;
```

Table 2 Query cases

Case	Platform	XPath expression
XM1	XMark	//open_auctions/open_auction//time
XM2	XMark	//people/person[.//address/city][.//creditcard]//name
XM3	XMark	//categories[./category[./name]/@id]//description
XM4	XMark	//categories/category[.//description//text[.//*//keyword]/bold]//name[contains(text(),'er')]
DB1	DBLP	//proceedings//url
DB2	DBLP	//article[.//url][./ee]//journal
DB3	DBLP	//inproceedings[.//title[./sub]//i]//url
DB4	DBLP	//phdthesis[./note/@type][.//url][./year > 2000]//author [contains(text(),'Michael')]

Proof In the dXML method, the fragment tree stores the complete node relation information from the root node to the leaf node, so that the collection of evaluation results of a branch path Q_p in all fragments S_p meets the $S_p \supseteq S_{p0}$ relationship with the evaluation results S_{p0} of Q_p in the original document tree. When S_p is de-duplicated, there is $S_p = S_{p0}$. The predicate filter conditions in the dXML method come from the collection of the evaluation results of each fragment and each branch path, and are recorded according to the INnodeFts. The filter condition obtained by Algorithm 7 is the complete judgment criterion for the retention of INnodeFt, so that the INnodeFt that can obtain the non empty final evaluation result is not omitted, that is, it meets the sufficient condition for obtaining the complete evaluation result of the INnodeFt. □

An example is given below to illustrate the process of obtaining predicate filter conditions. Suppose that the XML data to be queried is Fig. 3a and is partitioned as shown in the figure. If the XPath query is "/A[//B[/A//D][//E]/C]//D", the query contains complex nested multiple predicates, and the corresponding query tree is shown in Fig. 6. The nodes with gray background color in the figure are predicate items and the node with circle are return item. The branch paths contained in the query include ①/A//B/A//D, ②/A//B//E, ③/A//B/C and ④/A//D. Each branch path is represented by a number. The branch path ④ is the main path. The process is as follows:

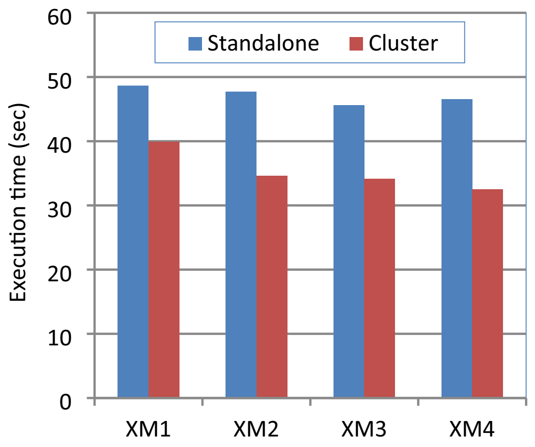
(1) Obtain local filter conditions

This step is completed on each worker. On each fragment, the filter conditions of each branch path are obtained respectively. First, record the nodes to be filtered. The nodes to be filtered of the three fragments are {A1, B1}, {A1, B1, A2, B2} and {A1, A2, B2}. Then, compared with the INnode list {A1, B1, A2, B2} obtained during XML parsing, the INnodeFts of each fragment can be obtained as {A1, B1}, {A1, B1, A2, B2} and {A1, A2, B2}. Next, the processing results on each fragment are as follows:

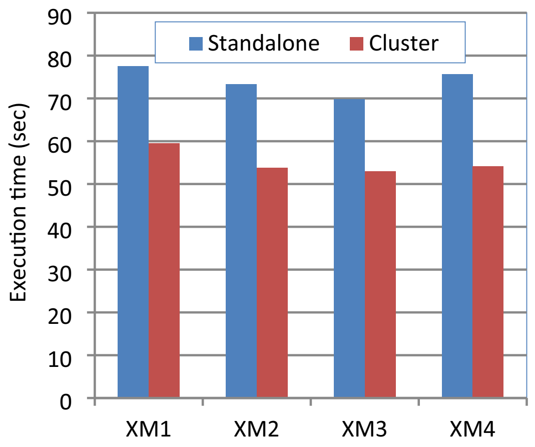
Evaluate on fragment I:	Evaluate on fragment II:	Evaluate on fragment III:
$\mathcal{V}_{\mathrm{I1}} = E(①) = \{\}$	$\mathcal{V}_{\mathrm{II1}} = E(①) = \{< B2,A1 >\}$	$\mathcal{V}_{\mathrm{III1}} = E(①) = \{\}$
$\mathcal{V}_{\mathrm{I2}} = E(②) = \{\}$	$\mathcal{V}_{\mathrm{II2}} = E(②) = \{\}$	$\mathcal{V}_{\mathrm{III2}} = E(②) = \{< B2,A1 >\}$
$\mathcal{V}_{\mathrm{I3}} = E(③) = \{< B1,A1 >\}$	$\mathcal{V}_{\mathrm{II3}} = E(③) = \{< B1,A1 >\}$	$\mathcal{V}_{\mathrm{III3}} = E(③) = \{< B2,A1 >\}$
$\mathcal{V}_{\mathrm{I4}} = E(④) = \{< D1,A1 >\}$	$\mathcal{V}_{\mathrm{II4}} = E(④) = \{< D2,A1 >\}$	$\mathcal{V}_{\mathrm{III4}} = E(④) = \{< D3,A1 >\}$

Table 3 Number of results of query cases

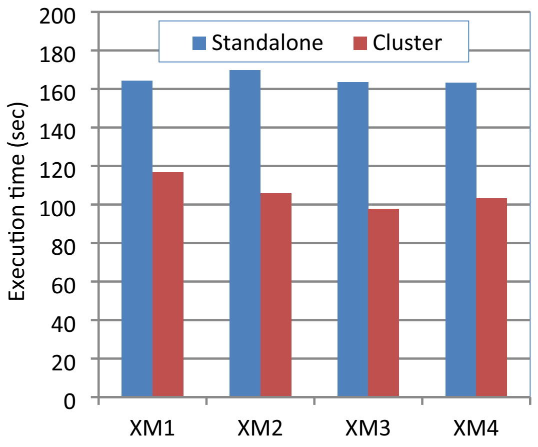
Case	Data size					
	1.25 GB	2.5 GB	5 GB	1.5 GB	2.25 GB	3 GB
XM1	658,184	1,258,596	2,574,541	–	–	–
XM2	70,318	134,398	275,015	–	–	–
XM3	11,000	21,000	43,000	–	–	–
XM4	275	536	1047	–	–	–
DB1	–	–	–	25,702	37,381	45,545
DB2	–	–	–	1,218,575	1,823,651	2,376,134
DB3	–	–	–	194	205	228
DB4	–	–	–	123	214	261



(a) 1.25GB



(b) 2.5GB



(c) 5GB

Fig. 7 Total execution time on XMark

The evaluation function *E* (*Pid*) is used to get the value chain set of the branch path *Pid* on the current fragment. The value chain set is represented by $\mathcal{V}$, and its subscripts are fragment id and branch path id. For example, the result sequence obtained by evaluating the branch path ③ on fragment I is $< C1, B1, A1 >$, where B1 and A1 are INnodeFts, while C1 is not INnodeFt, so the recorded value chain is $< B1, A1 >$.

(2) Collect filter conditions.

This step is completed on the coordinator. Its function is to collect the received filter conditions from each worker according to each branch path. The collecting process only records different value chains from each worker.

Collect for branch path ①: $\mathcal{V}_{c1}=\mathcal{V}_{I1}\cup\mathcal{V}_{II1}\cup\mathcal{V}_{III1}=\{<B2,A1>\}$

Collect for branch path ②: $\mathcal{V}_{c2} = \mathcal{V}_{I2} \cup \mathcal{V}_{II2} \cup \mathcal{V}_{III2} = \{< B2,A1 >\}$.

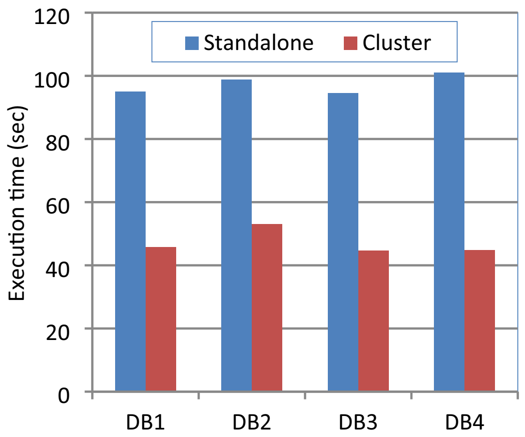
Collect for branch path ③: $\mathcal{V}_{c3} = \mathcal{V}_{I3} \cup \mathcal{V}_{II3} \cup \mathcal{V}_{III3} = \{< B1,A1 > , < B2,A1 >\}$.

Collect for branch path ④: $\mathcal{V}_{c4} = \mathcal{V}_{I4} \cup \mathcal{V}_{II4} \cup \mathcal{V}_{III4} = \{< D1,A1 > , < D2,A1 > , < D3,A1 >\}$.

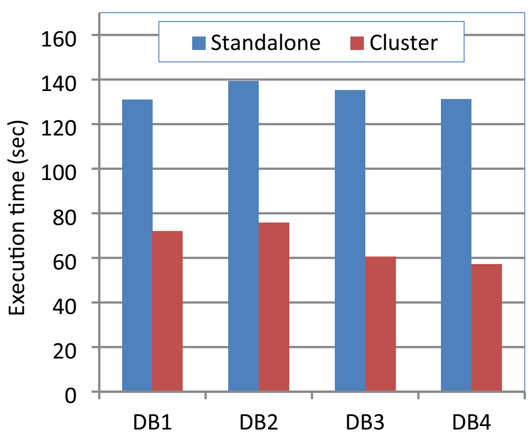
The basic collecting operation $\cup$ is the union of sets of value chains.

(3) Merge filter conditions

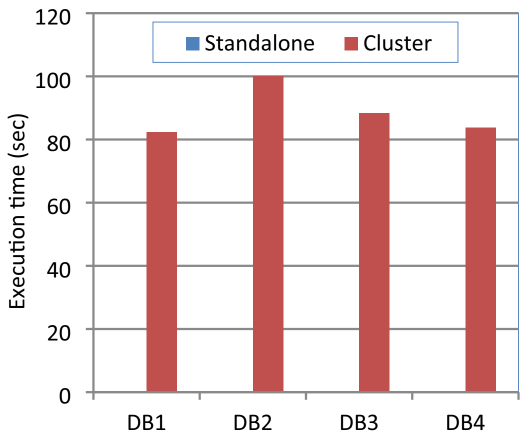
This step continues on the coordinator. Merging is performed for the nodes to be filtered by predicates, from the last level to the first level, to process the collection results of filter conditions. The basic merge operation $\cap_P$ takes the predicate node to be filtered *P* as the merging point to obtain the common part of the value chain set, that is, to carry out the intersection of the value chain set. In this example,



(a) 1.5GB



(b) 2.25GB



(c) 3GB

Fig. 8 Total execution time on DBLP

the predicate evaluation includes B[] and A[] in order from the last level to the first level, so there are:

Merge for B[]:$\mathcal{V}_{\text{mB}} = \mathcal{V}_{\text{c1}} \cap_{\text{B}} \mathcal{V}_{\text{c4}} \cap_{\text{B}} \mathcal{V}_{\text{c3}} = \{< \text{B2,A1} >\}$.

Merge for A[]:$\mathcal{V}_{\text{ret}} = \mathcal{V}_{\text{mB}} \cap_{\text{A}} \mathcal{V}_{\text{c4}} = \{< \text{A1} >\}$.

Since the pre evaluating result of this example is {D1, D2, D3}, the query result is {D1, D2, D3} after filtering the pre evaluating result according to the returned final filter condition $\mathcal{V}_{\text{ret}}$.

4 Experiments

4.1 Experimental Settings

We use two typical common test platforms to carry out the experiment. One is the XMark [49] platform, which provides a tool for generating XML data of any size. We use XM1 ~ XM2 cases in Table 2 for testing, of which XM1 is a simple linear path query, XM2 is a query with multiple juxtaposed predicates, XM3 is a query with nested multiple predicates and a query step to obtain attribute nodes, and XM4 is a complex query with not only nested multiple predicates but also wildcard query and function call. The other is the DBLP platform [3], which provides real data of computer science bibliographies described in XML. The tests were performed using DB1 ~ DB4 cases in Table 1, covering simple path queries and complex multi-predicate queries. DB1 is a simple path query, DB2 has juxtaposed multiple predicates, DB3 has nested predicates, and DB4 has function calls and content comparison operation.

The hardware platform is a high-performance server with the model of HP ProLiant DL380. Its hardware configuration includes two Intel Xeon E5-2660 CPUs, which can provide 40 CPU threads, a total capacity of 164 GB memory and a total capacity of 4.8 TB hard disk space. By configuring VMware ESXi 5.5 software, a small cluster of

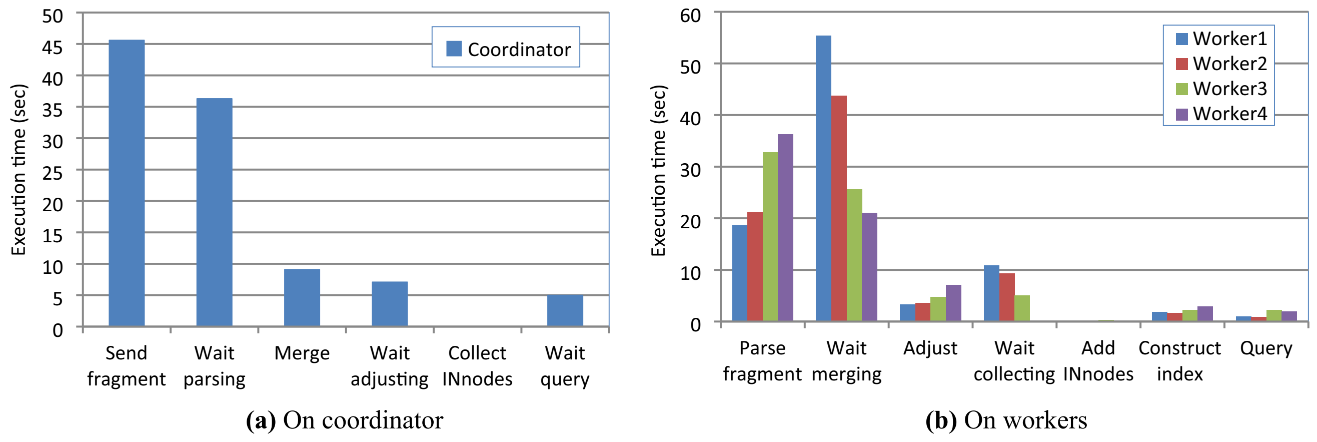


Fig. 9 Execution time of each step on coordinator and workers (case XM4, 5 GB)

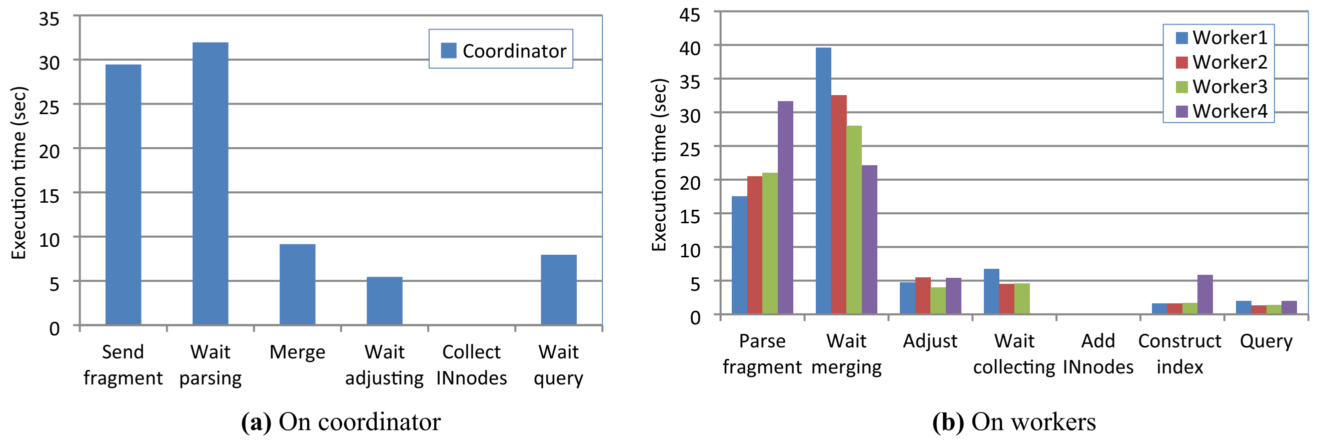


Fig. 10 Execution time of each step on coordinator and workers (case DB4, 3 GB)

1 coordinator and 4 workers is obtained. The coordinator and each worker are configured with four CPU threads; 32 GB of memory and 300 GB of disk space. The type of virtual network adapter is E1000, which simulates a gigabyte network. Both the coordinator and the worker run Centos 7.6 operating system, and the Java virtual machine environment used for software development and running is JRE1.8.

4.2 Performance evaluation

To assess the overall performance of dXML method, different sizes of XML data and various typical XML queries are tested. The total execution time of dXML in cluster is investigated and compared with that of standalone program. In the cluster, since users submit cluster processing requests to the coordinator through the client, the total execution time of distributed computing is the time from receiving the query program on the coordinator to the return of the final query result. For the fairness of comparison, the hardware and software environment of the standalone program is consistent with that of the worker in the cluster, and the standalone program adopts the same data model and basic evaluation method as that in dXML method. The configuration of XML data size is shown in Table 3. The data in the table is the number of results that can be obtained by query. The data provided for XMark queries is generated data with data size of 1.25 Gb, 2.5 Gb and 5 GB respectively, and the magnification of data size is 1, 2 and 4 respectively. DBLP queries are provided with real data with data size of 1.5 GB, 2.2 GB and 3 GB respectively, and the magnification is 1, 1.5 and 2 respectively.

The experimental results of the total execution time are shown in Figs. 7 and 8, which correspond to the queries on XMark platform and the queries on DBLP platform respectively. Each sub graph shows the execution results under different data size conditions. In Fig. 8c, when the

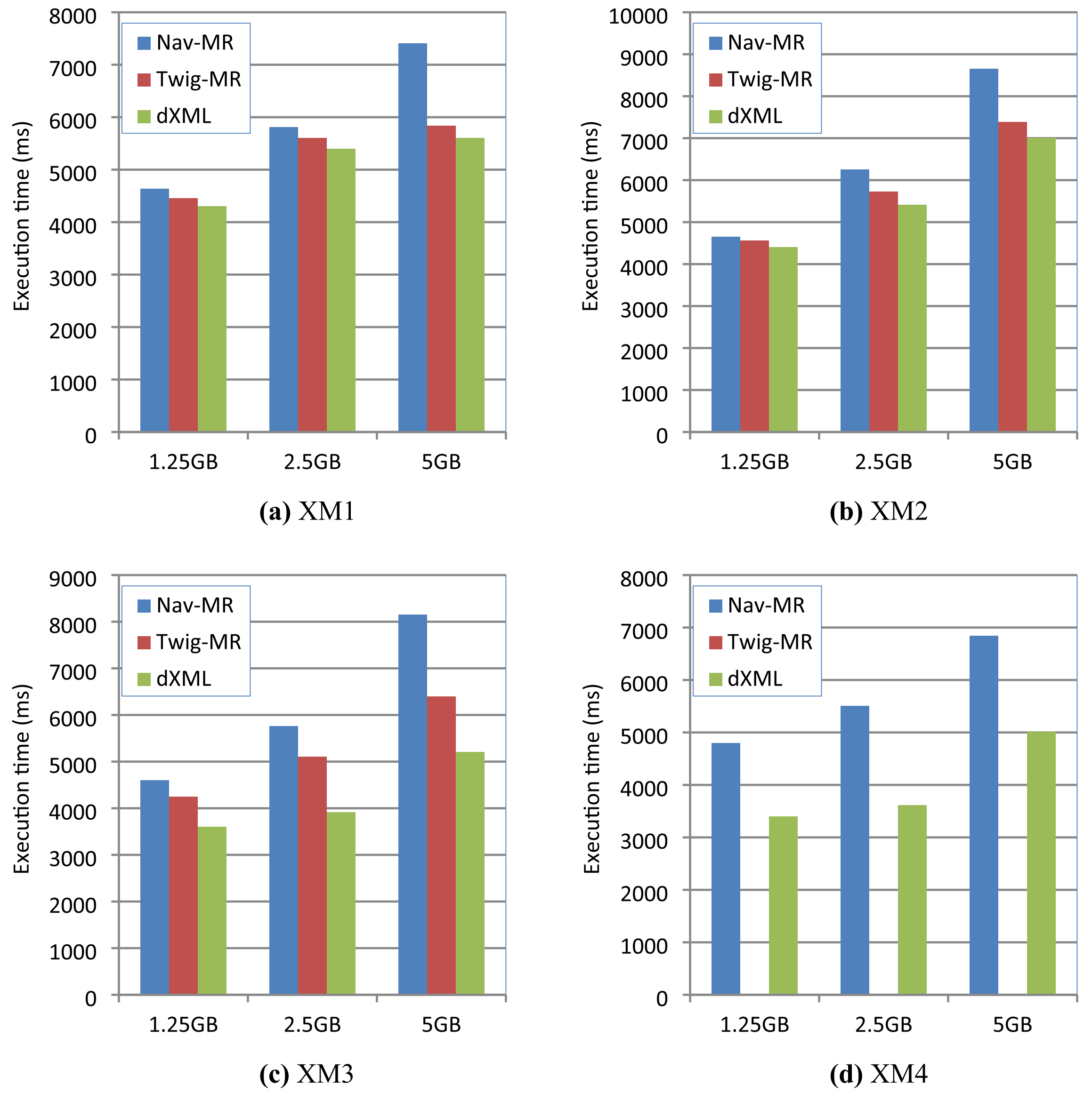


Fig. 11 Comparison of query time on XMark

data size is 3 GB, the standalone execution time is not presented, the reason is that in standalone manner, each query cannot return the final result due to timeout.

It can be seen from the figure that the performance of dXML in cluster manner is better than that of standalone program. The extent of performance improvement is related to specific queries and XML data. Taking the execution speed as the performance indicator, the performance improvement refers to the average improvement of the execution speed of each case in cluster manner compared with that in standalone manner under various test conditions (specified query and data size). Equations 1 and 2 are used to calculate the performance improvement. Equation 1 is used to calculate the performance improvement of a single case, where T_{si} and T_{ci} are the execution time in the standalone and cluster manner respectively. Equation 2 is used to calculate the average performance improvement of all cases. On XMark platform, the performance of cluster manner is 41% higher than that of standalone manner on average, while on DBLP platform, the performance improvement is up to 104%. In addition, when the data size increases to a certain value, the final result cannot be obtained due to timeout in standalone manner. It shows that the use of distributed computing can break through the dilemma that a single machine cannot process large data due to its limited processing capacity.

$$g_i = (1/T_{ci} - 1/T_{si})/(1/T_{si}) = T_{si}/T_{ci} - 1 \tag{1}$$

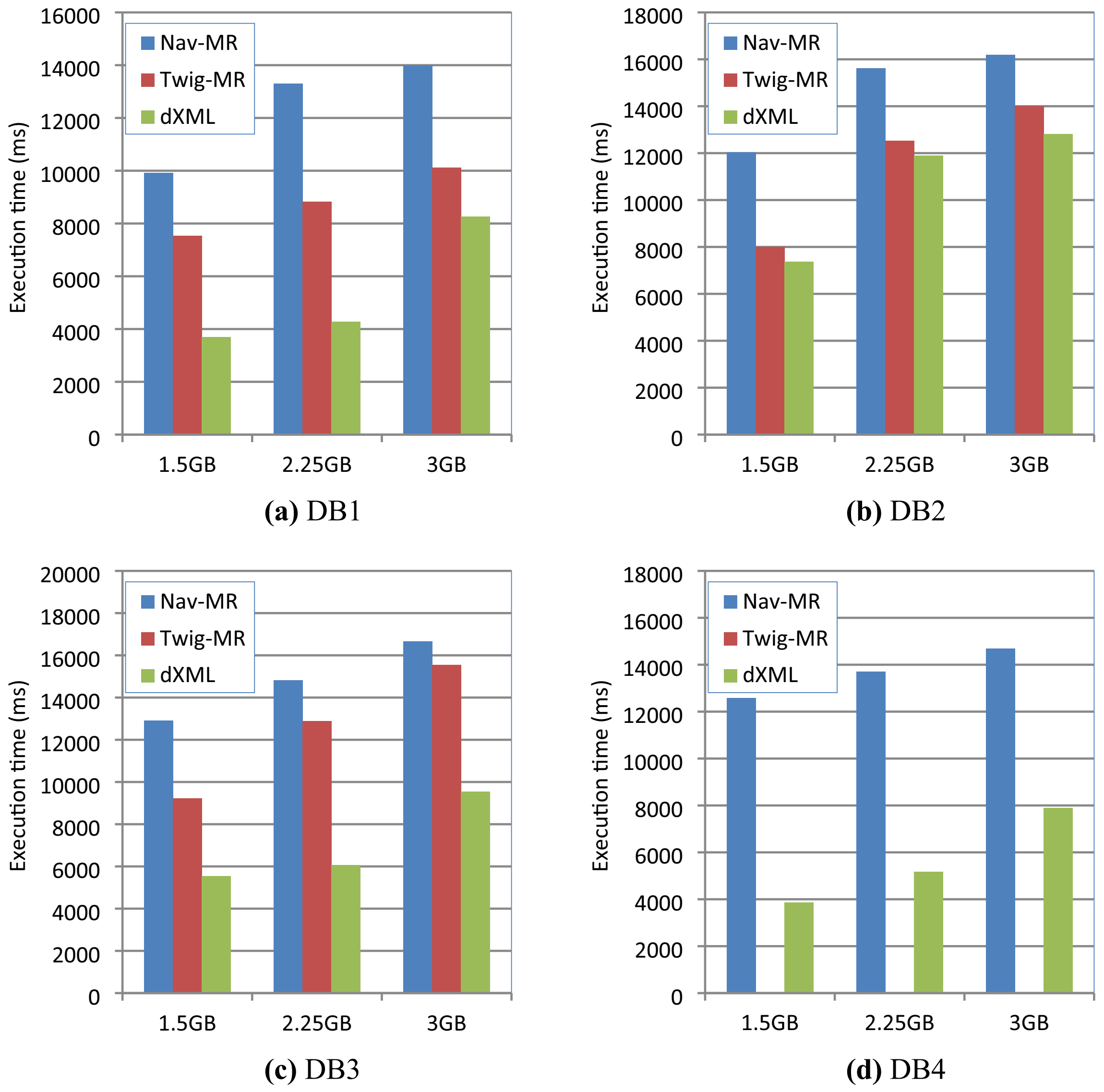


Fig. 12 Comparison of query time on DBLP

$$\overline{g} = \sum_{i=1}^{n} g_i / n \tag{2}$$

In order to further investigate the execution of each step in dXML method, we recorded the execution time of each step in the process of dXML. Since the distributed processing of dXML includes the execution on the coordinator and the execution on each worker, we show it respectively according to the coordinator and worker. Figures 9 and 10 respectively illustrate the execution of case XM4 on XMark platform when the data size is 5 GB and case DB4 on DBLP platform when the data size is 3 GB. The results show that the execution time of the distributed data preparation phase accounts for a large proportion, while the time of the query phase accounts for a relatively small proportion. In specific processing steps, such as parsing on fragment, due to the unbalanced load on each worker, it directly caused a long time of synchronous waiting.

4.3 Comparative experiments

Current related methods mainly focus on the efficiency of the XML query part, while XML data is usually partitioned and parsed serially in the data preparation phase prior to XML query. In addition, the data preparation phase is time-consuming for any method, but has no specificity for related method. The difference between various XML processing methods is mainly reflected in the query approach. Therefore, the comparative experiment in this section is limited to the query stage, and takes the evaluation efficiency of XPath as the comparison object. Because the steps of XML parsing, index creation and XPath query are relevant in XML processing, for the sake of fairness, the data to be queried used in each method of comparison are consistent. We choose two kinds of typical methods for comparison. One is navigational method, and HoX-MaRe [35] is selected for comparison; the other is twig method, and TwigStack-MR [12, 43] method is selected for comparison. In this paper, the implementation versions of these two comparative methods are named Nav-MR and Twig-MR respectively. Both methods use hadoop-2.8.5 as the development and running environment. In the testing process of each method, the XML data is parsed in advance, the data to be queried is in a ready state, and the execution time of the query stage is recorded. For Nav-MR method, the parsed data is pre-partitioned into horizontal fragments. The query time includes the navigational evaluation time on each XML fragment in MAP stage and the result merging time in Reduce stage. For the Twig-MR method, the query time includes the time to query the XML subtrees using the TwigStack method in Map stage and the merging time of the results in Reduce stage. Since the construction of the subtrees has been completed in advance and is not in the process of querying, the lower bound of query time for Twig-MR is obtained. The query time of dXML includes the query time on the worker and the merging time on the coordinator. The actual measurement is the whole period from the query rewriting on the coordinator to the return of the final result. The test cases adopt the same configuration as Sect. 4.2. The query time comparison results on XMark and DBLP platforms are shown in Figs. 11 and 12 respectively. For case XM4 and case DB4, the twig method does not support query semantics with function call, so there are no corresponding results in Figs. 11d and 12d.

In terms of query execution time, the performance of dXML is better than other methods based on MapReduce. This is mainly due to two factors: One is the basic performance of the algorithm. dXML adopts the query method based on relation index, which can carry out efficient navigational query. Another is that in terms of distributed computing framework, dXML adopts a flexible algorithm framework different from MapReduce. MapReduce is suitable for batch processing of large-scale data, but the complex XML processing process needs to be transformed into multiple batches, and the performance is negatively affected. In addition, the processing method based on MapReduce has the problem of data replication [50]. The high replication rate increases the network traffic, resulting in limited overall performance. By contrast, dXML provides a controllable distributed XML processing method. Since dXML supports flexible parsing of arbitrary XML fragments, it is conducive to load balancing. In the query stage, the filter-upon-pre-evaluate approach can obtain better data locality and avoid excessive communication overhead. In addition, because dXML supports XML fragment parsing and independent index creation, it can make full use of the distributed computing environment for parallel processing.

5 Discussion

Although dXML can effectively utilize distributed computing resources for large XML data processing, there is still a need for performance optimization. From the performance evaluation experiments in Sect. 4.2, the results of the time consumption of each execution step in dXML show that the current performance limitations come mainly from the following three aspects: first, the unbalanced load on each worker; second, the long fragment sending time; and third, the high percentage of XML parsing time. Therefore, the corresponding improvement measures include:

(1) Load balance on each worker. Due to the unbalanced load, synchronous waiting has become an important factor causing the performance bottleneck. In the current test, XML data is only evenly partitioned according to the data size, which is easy to cause load imbalance in local parsing and query. Fortunately, since dXML supports the processing of arbitrary fragments of XML data, it is easy to combine with a more optimized load balancing strategy.

(2) Network performance. Network performance is another important factor that affects dXML efficiency. Optimizing network communication conditions in the cluster can improve overall performance. Because dXML does not rely on any specific computing framework, it is highly adaptable to changes in the computing environment.

(3) Preprocessed data loading. Optimize preprocessed data loading by reusing parsing results and indexes. Because the parsing and indexing of XML data account for a large proportion in the whole XML processing process, the parsed data and created indexes can be reused to improve the performance of multiple queries in practical application scenarios.

6 Conclusion

Due to the limitation of standalone computing capacity, it is an inevitable choice to use distributed computing to process large-scale XML data in the cluster. The dXML method proposed in this paper is an integrated XML processing technology including distributed parsing, distributed indexing and distributed query of XML data. Our method supports the distributed parsing of arbitrary XML fragments and the distributed creation of indexes. Distributed XPath evaluation is carried out through the filter-upon-pre-evaluate approach, which realizes the data locality during query and reduces the network traffic. In general, our method is an integrated automatic distributed processing of XML, and can support ad hoc query of large XML data. The working mechanism of dXML has strong flexibility, does not rely on the existing distributed framework, and does not need a complex environment configuration. Therefore, it is a lightweight solution. Through experimental evaluation, we found that under various query conditions, the performance of dXML in cluster manner is superior to that in standalone manner, and can overcome the dilemma that single machine can not handle large XML data. Through comparative experiments, it shows that dXML has advantages over the existing typical methods based on MapReduce in terms of XML query performance. Based on the discussion in Sect. 5, our future work focuses on performance optimization of dXML. The measures considered include optimization of the load balancing approach, optimization of cluster network performance, and optimization of preprocessed data loading. In addition, effective automatic processing mechanisms need to be provided for these optimizations.

Author contributions RC: conceptualization, formal analysis, software, wrote original draft. GC: validation, project administration. JC: validation, software. YH: validation, reviewed & edited.

Funding This research was supported by the Natural Science Foundation of Fujian Province of China (2022J01336, 2022J01820) and Open Fund of Digital Fujian Big Data Modeling and Intelligent Computing Institute.

Data availability Enquiries about data availability should be directed to the authors.

Declarations

Conflict of interest There is no conflict of interest by any of the authors of this article.

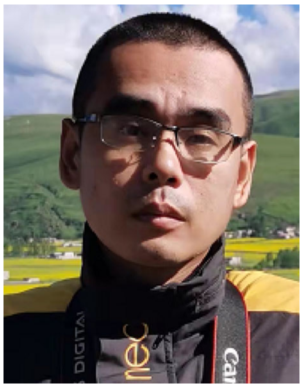
References

1. Zhen, H.L., Murthy, R.: A Decade of XML data management: an industrial experience report from oracle. In: Proceedings of the 25th International Conference on Data Engineering (ICDE 2009), Shanghai, China, March 29 - April 2 2009 2009, pp. 1351–1362. IEEE Computer Society
2. Lee, H.: Data storage practices and query processing in XML databases: a survey. Knowl.-Based Syst. **24**(8), 1317–1340 (2011)
3. DBLP XML dataset. http://dblp.uni-trier.de/xml/.
4. Wikimedia XML dataset. http://download.wikimedia.org/enwiki/latest.
5. OpenStreetMap XML dataset. http://www.openstreetmap.org/export.
6. Sankari, S., Bose, S.: Elaborative survey on storage technologies for XML big data: A real-time approach. In: 2016 International Conference on Recent Trends in Information Technology (ICRTIT) 2016
7. Brahmia, Z., Hamrouni, H., Bouaziz, R.: XML data manipulation in conventional and temporal XML databases: a survey. Comput. Sci. Rev. **36**, 100231 (2020)
8. Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J., Siméon, J.: XML path language (XPath) 2.0 (Second Edition). W3C recommendation (2015).
9. Dean, J.: MapReduce : simplified data processing on large clusters. In: Symposium on Operating System Design & Implementation 2004
10. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. ACM SIGMOD Rec. **40**(4), 11–20 (2012)
11. Gou, G., Chirkova, R.: Efficiently querying large XML data repositories: a survey. IEEE Trans. Knowl. Data Eng. **19**(10), 1381–1403 (2007)
12. Fan, H., Ma, Z., Wang, D., Liu, J.: Handling distributed XML queries over large XML data based on MapReduce framework. Inform. Sci. **15**, 2–89 (2018)
13. Chen, R., Liao, H.: ParaParse: A parallel method for XML parsing. In: Proceedings of the 3rd IEEE International Conference on Communication Software and Networks (ICCSN2011) 2011, pp. 81–85
14. Chen, R., Liao, H., Wang, Z.: Parallel XPath evaluation based on node relation matrix. J. Comput. Inform. Syst. **9**(19), 7583–7592 (2013)
15. Chen, R., Wang, Z., Su, H., Xie, S., Wang, Z.: Parallel XPath query based on cost optimization. J. Supercomput. (2021). https://doi.org/10.1007/s11227-021-04074-y
16. Cate, B.T., Marx, M.: Navigational XPath. ACM. SIGMOD Record **36**(2), 19–26 (2007)
17. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3–6, 2002 2002, pp. 310–321. ACM
18. Lukas, P., Baca, R., Kratky, M., Ling, T.W.: Demythization of structural XML query processing: comparison of holistic and binary approaches. IEEE Trans. Knowl. Data Eng. **33**(04), 1439–1452 (2021)
19. Sato, S., Hao, W., Matsuzaki, K.: Parallelization of XPath Queries Using Modern XQuery Processors. In: New Trends in Databases and Information Systems. ADBIS 2018 2018 (2018)
20. Mortier, R., Narayanan, D., Donnelly, A., Rowstron, A.: Seaweed: Distributed Scalable Ad Hoc Querying. In: International Conference on Data Engineering Workshops 2006
21. White, T.: Hadoop: the definitive guide. O'rlly Media Inc Gravenstn Highway North **215**(11), 1–4 (2012)
22. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. (2010).
23. 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: Acm International Conference on Information & Knowledge Management 2012

24. Owen, S., Kwon, H.: Spark-XML. https://github.com/databricks/spark-xml (2015).
25. Bidoit, N., Colazzo, D., Sartiani, C., Solimando, A., Ulliana, F.: Andromeda: a system for processing queries and updates on big XML documents. In: East European Conference on Advances in Databases & Information Systems 2015
26. Bidoit, N., Colazzo, D., Malla, N., Sartiani, C.: Evaluating Queries and Updates on Big XML Documents. Inf. Syst. Front. **20**(1), 63–90 (2018)
27. Camacho-Rodriguez, J., Colazzo, D., Manolescu, I.: PAXQuery: Efficient Parallel Processing of Complex XQuery. IEEE Trans. Knowl. Data Eng. **27**(7), 1–1 (2015)
28. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J., Stefanescu, M.: XQuery 1.0: An XML query language (Second Edition). W3C working draft (2010).
29. Carman, E.P., Westmann, T., Borkar, V.R., Carey, M.J., Tsotras, V.J.: A scalable parallel XQuery processor. In: IEEE International Conference on Big Data 2015
30. Using Oracle XQuery for Hadoop. http://docs.oracle.com/cd/E63064_01/doc.42/e63063/oxh.htm #BDCUG527 (2016).
31. Hricov, R., Šenk, A., Kroha, P., Valenta, M.: Evaluation of XPath Queries Over XML Documents Using SparkSQL Framework. In: International Conference: Beyond Databases, Architectures and Structures 2017
32. Khatchadourian, S., Mariano P. Consens, Siméon, J.: Having a ChuQL at XML on the Cloud. In: AMW 2011
33. Fegaras, L., Philip, J.J.: XML Query Optimization in Map-Reduce. In: International Workshop on the Web & Databases 2011
34. Senk, A., Valenta, M., Benn, W.: Distributed Evaluation of XPath Axes Queries over Large XML Documents Stored in MapReduce Clusters Paper presented at the DEXA.2014,
35. Damigos, M., Gergatsoulis, M., Plitsos, S.: Distributed Processing of XPath Queries Using MapReduce. (2014).
36. Kunfang, S., Lu, H.: Efficient querying distributed big-XML Data using MapReduce. Int. J. Grid High Perf. Comput. **8**(3), 70–79 (2016)
37. Liang, B.A., Jin, Y.A., Cqw, B., Hq, A., Xin, Z.A., Sc, A.: XML2HBase: Storing and querying large collections of XML documents using a NoSQL database system. J. Parall. Distrib.Comput. **161**, 83–99 (2021)
38. Apache HBase. https://hbase.apache.org/.
39. Liu, J., Liu, Q., Zhang, L., Su, S., Liu, Y.: Enabling massive XML-based biological data management in HBase. IEEE/ACM Trans. Comput. Biol. Bioinf. **17**(6), 1994–2004 (2020)
40. Longjian, Y., Koide, H., Cavendish, D., Sakurai, K.: Efficient Shortest Path Routing Algorithms for Distributed XML Processing. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies 2019, pp. 265–272
41. Bi, X., Zhao, X.G., Wang, G.R.: Efficient processing of distributed twig queries based on node distribution. J. Comput. Sci. Technol. **32**(1), 78–92 (2017)
42. Subramaniam, S., Haw, S.C., Soon, L.K.: Improved centralized XML query processing using distributed query workload. IEEE Access **9**, 29127–29142 (2021)
43. Fan, H., Yang, H., Ma, Z., Liu, J.: TwigStack-MR: An Approach to Distributed XML Twig Query Using MapReduce. In: IEEE International Congress on Big Data 2016, pp. 133–140
44. Braganholo, V., Mattoso, M.: A Survey on XML Fragmentation. ACM SIGMOD Record (2014).
45. Choi, H., Lee, K.H., Lee, Y.J.: Parallel labeling of massive XML data with MapReduce. J. Supercomput. **67**(2), 408–437 (2014)
46. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On supporting containment queries in relational database management systems. In: ACM SIGMOD Record, 2001 2001, vol. 2, pp. 425–436. ACM
47. Lu, J., Meng, X., Ling, T.W.: Indexing and querying XML using extended Dewey labeling scheme. Data Knowl. Eng. **70**(1), 35–59 (2011)
48. Hsu, W.-C., Shih, H.-C., Liao, I.-E.: A scalable XML indexing method using MapReduce. In: Fourth edition of the International Conference on the Innovative Computing Technology (INTECH 2014) 2014, pp. 81–86. IEEE
49. Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: Proceedings of the 28th international conference on Very Large Data Bases 2002, pp. 974–985. VLDB Endowment
50. Afrati, F., Damigos, M., Gergatsoulis, M.: Lower bounds on the communication of XPath queries in MapReduce. (2015).

Rongxin Chen received the Ph.D. degree in computer science from Beijing University of Technology, Beijing, China, in 2014. He is currently a professor with Computer Engineering College, Jimei University, Xiamen, China. He is also a researcher with Digital Fujian Big Data Modeling and Intelligent Computing Institute, Xiamen, China. His research interests include parallel and distributed computing, data processing and analysis technology.

Guorong Cai received the Ph.D. degree in computer science from Xiamen University, Xiamen, China, in 2013. He is currently a professor with Computer Engineering College, Jimei University, Xiamen, China. His research interests include machine learning, pattern recognition, and high performance computing.



Jie Chen received the M.S. degree in mathematics from Fujian Normal University, Fuzhou, China, in 2004. He is currently a senior lecturer with Computer Engineering College, Jimei University, Xiamen, China. His research interests include program analysis and algorithm design.

Yuling Hong received the Ph.D. degree in systems engineering from Fuzhou University, Fuzhou, China, in 2020. She is currently an associate professor with Computer Engineering College, Jimei University, Xiamen, China. Her research interests include network big data mining, parallel and distributed computing.