



Handling distributed XML queries over large XML data based on MapReduce framework



Hongjie Fan^{a,b}, Zhiyi Ma^{a,b,*}, Dianhui Wang^c, Junfei Liu^d

^a School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

^b Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China

^c Department of Computer Science and Information Technology, La Trobe University, Melbourne, VIC 3086, Australia

^d National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China

ARTICLE INFO

Article history:

Received 11 September 2017

Revised 3 April 2018

Accepted 5 April 2018

Available online 11 April 2018

Keywords:

XML

XPath query

Twig query

Hadoop

MapReduce

ABSTRACT

With the increase in available extensible markup language (XML) documents, numerous approaches to querying have been proposed in the literature. XPath queries and Twig pattern queries are the two basic approaches, directly affecting the efficiency of XML operations. Distributive manipulation of massive XML data is challenging. This paper aims to develop an efficient distributed XML query processing method using MapReduce, which simultaneously processes several queries on large volumes of XML data. First, we split up a large-scale XML data file into file-splits and put them in a distributed storage system. Then, we present an efficient algorithm to compute different fragments of the document tree using the MapReduce framework in parallel. In order to efficiently handle a large amount of XML data, we built a partition index and used a random access mechanism for specific queries. The experiment results show that our proposed approach is efficient with good scalability.

© 2018 Elsevier Inc. All rights reserved.

1. Introduction

The extensible markup language (XML)¹ has become the de facto standard for data storage on the World Wide Web, where the amount of data is constantly increasing. Wikipedia alone is an XML document with a size of over 40 GB.² UniProtKB,³ the largest scientific XML document that houses a collection of functional entries and relationships between proteins, now has more than 259 GB of data. The rapid growth of XML data increases the demand for methods for the distributed processing of XML queries. XPath⁴ and Twig [1] queries are two basic XML queries, directly affecting the efficiency of XML operations. The XPath querying method, as a core subset of the XML query process, can be used to retrieve a subset of the XML data nodes that satisfy some path constraints. The Twig querying method uses a small query tree, and is essentially a complex selection performed on the structure of an XML document.

* Corresponding author at: School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China.

E-mail addresses: hjfan@pku.edu.cn (H. Fan), mazhiyi@pku.edu.cn (Z. Ma), dh.wang@latrobe.edu.au (D. Wang), liujunfei@pku.edu.cn (J. Liu).

¹ <https://en.wikipedia.org/wiki/XML>.

² http://en.wikipedia.org/wiki/Wikipedia:Database_download.

³ <http://www.uniprot.org/>.

⁴ <http://www.w3.org/TR/xpath20/>.

Traditional XML query methods cannot be efficiently used to manipulate massive XML datasets, especially in a distributed manner. This is because conventional queries are mainly devised to process a series of small-size documents and are not suitable for large-size XML files. Current research on XML query processing algorithms that use cloud computing are at a very preliminary stage. Processing queries on XML documents can be done quite efficiently in centralized environments, but traditional algorithms are not designed to evaluate queries on massive documents. The exploration of massive XML data still remains challenging due to the following reasons:

1. A massive XML document has complex internal structures and semantic information, which is difficult to split into partitions for a distributed storage system.

2. It is challenging to design distributed XML queries such as XPath and Twig queries in a distributed storage system.

To address these issues, we consider the problem of handling distributed XML queries over large XML datasets in this paper, aiming to speed up the process within a decentralized structure, which amounts to developing a data partitioning strategy and a model for parallel computing. For example, given a document, first we parse it based on a simple API for XML (SAX).⁵ Then, we split up the obtained XML data into file-splits. Consequently, the system is ready to process XML queries in a parallel manner. Hadoop⁶ is an open source implementation of a parallel processing paradigm known as MapReduce [2]. A distributed platform such as Hadoop,⁶ which supports query processing, can be regarded as a model. It is suitable for querying and has demonstrated excellent performance. Users can design algorithms that comply with the Hadoop framework characteristics, which ensures the correctness and completeness of the approach. The parallel algorithm is executed as a MapReduce [2] task on the Hadoop platform. This is widely acknowledged as an effective model for querying on large scale datasets. During the Map step, clusters process intermediate results, while the task of Reduce will generate the final result. Experiments show that the proposed approach is efficient and sufficiently scalable.

This work is built on our previous studies reported in [3,4], where two types of XML queries were considered separately. To unify our proposed retrieval techniques for large-scale XML datasets, we further develop the existing framework with a consistent workflow. The technical contributions from this work can be summarized as follows:

1. A consistent XML document segmentation method for more general queries is proposed in a distributed environment.
2. Two methods are proposed for splitting massive XML data into file-splits as inputs for a distributed storage system.
3. A MapReduce framework is designed for distributed XML queries, XPath and Twig queries, on large-scale XML datasets.
4. A Hadoop and MapReduce-based method of distributed querying is presented.
5. The index on XPath queries is suggested to further improving of the overall performance.
6. Experiments using distributed querying were conducted. The results show that the proposed approach is efficient and sufficiently scalable.

The remainder of the paper is organized as follows: Section 2 provides the background and preliminaries related to XML, XPath query, Twig query, and the MapReduce workflow. Section 3 describes the parallel query method, including the process of splitting XML data using virtual nodes or fragments of certain types, the distributed XPath query, and the parallel TwigStack-MR algorithm. Section 4 reports our experimental results, and Section 5 concludes this paper with some directions for future work.

2. Background and preliminaries

To keep this paper self-contained, we briefly review some relevant concepts, such as XML, the XPath query, the Twig query, the Hadoop platform, and the MapReduce framework. Readers can refer to [3,4] for more details.

2.1. The XML database and storage scheme representation

An XML database is built by a collection of XML documents.¹ In the current work, the considered database is XML of a large-scale one. Fig. 1 shows that XML is similar to HTML, except that XML has stricter rules concerning opening and closing tags. This means that once a tag is opened (for example `<book>`), it has to be closed (using `</book>`).

Although XML documents can have rather complex internal structures, we can model them as trees. For example, in Fig. 2, each pair of tags is a node in the document tree. In the XML *L-Stream Storage Scheme* [5], also called the region encoding labeling scheme, the position of a string or element in the XML database is represented as a 3-tuple (*DocId*, *StartPos*:*EndPos*, *LevelNum*), where *DocId* is the identifier of the document; *StartPos* and *EndPos* can be generated by counting word numbers from the beginning of the document with identifier *DocId* until the start of the element and end of the element, respectively, which are assigned according to an order given by a depth first search (DFS) variation. *LevelNum* is the nesting depth of the element (or string value) in the document. In this paper, we take the dataset as a single document, so we omitted *DocId* and adopted a tuple (*StartPos*:*EndPos*, *LevelNum*) instead.

In the XML *L-Stream Storage Scheme*, there are four types of relationships between node *x* and *y*, which are listed in Table 1. A Parent-Child (P-C) relationship between node *x* and *y* exists if $x.StartPos < y.EndPos$ ($x.SP < y.EP$) AND $x.EndPos < y.EndPos$ ($x.EP < y.EP$). Meanwhile, between nodes *x* and *y* the level number satisfies $y.LevelNum = x.LevelNum + 1$. An

⁵ <http://www.saxproject.org/>.

⁶ <http://hadoop.apache.org/>.

```

<book>
  <title>XML</title>
  <chapter>
    <section>
      <head>introduction</head>
      <subsection>data model</subsection>
    </section>
    <section>
      <head>tree pattern</head>
      <subsection>twig</subsection>
    </section>
    <section>
      <head>search</head>
    </section>
  </chapter>
  <author>Lu</author>
  <year>2013</year>
</book>

```

Fig. 1. Fragment of a Sample XML document.

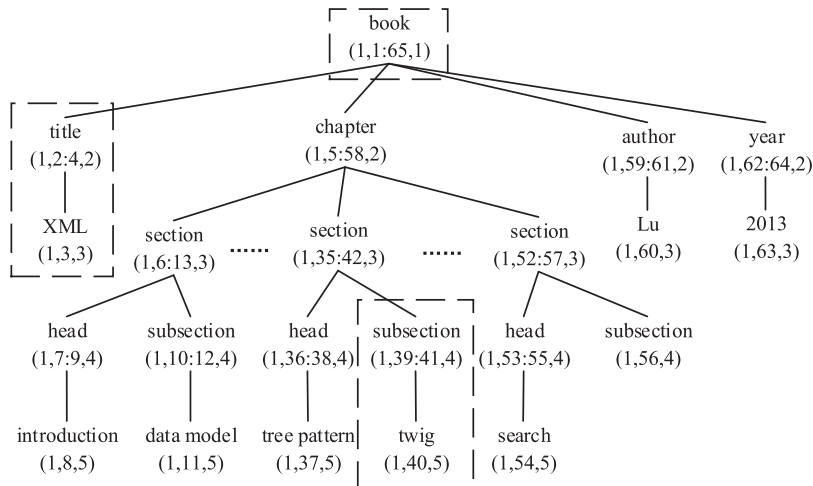


Fig. 2. Tree representation of an XML document fragment.

Ancestor-Descendant (A-D) relationship exists between node x and y if $x.StartPos < y.EndPos(x.SP < y.EP)$ AND $x.EndPos < y.EndPos(x.EP < y.EP)$.

2.2. The XPath and Twig query

XPath is an expression language that can process values conforming to the data model, which provides a tree representation of XML documents, as shown in Fig. 2. XPath is, fundamentally, a general purpose language for addressing, searching, and matching pieces of an XML document. The semantics of the language is based on a representation of the information content of an XML document as an ordered tree [6]. The result of an XPath expression may be a selection of nodes from the input XML documents.

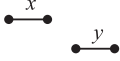



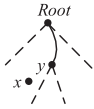

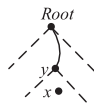
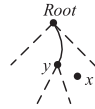
XQuery is a query languages defined by the W3C (World Wide Web Consortium)⁷ for querying XML documents. It is a declarative, statically-typed query language for querying collections of XML documents, a file system or a database [6]. It is based on the same interpretation of XML documents as XPath, but has the added ability to query multiple documents in a collection of XML documents and combine the results into completely new XML fragments [6]. An XQuery is essentially a complex selection on the structure of an XML document and can be used to locate element nodes on the data tree corresponding to the XML document of interest. It is deeply rooted in almost every XML technology, ranging from query languages to access control languages.

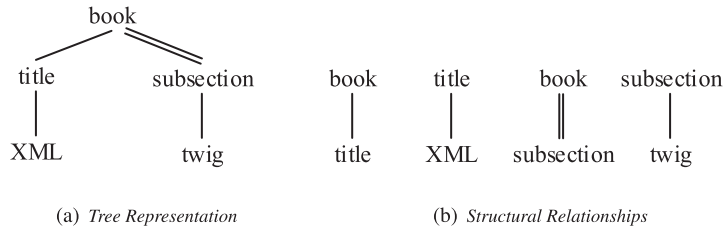
In [7], Bidoit et al. investigated the distributed evaluation of XPath and XQuery languages over a single document. In [8], Cong et al. proposed a distributed XML query-processing algorithm using a *Partial Evaluation* strategy. In [9], Choi et al.

⁷ <https://www.w3.org/>.

Table 1

Cases for the L-stream storage scheme.

	Case 1	Case 2	Case 3	Case 4
Property	$x.EP < y.SP$	$x.SP < y.EP$ and $x.EP < y.EP$	$x.SP < y.SP$ and $x.EP < y.EP$	$x.SP < y.EP$
Segment				
Tree				

**Fig. 3.** Twig query and structural relationships.

designed and presented a HadoopXML system for solving the problem of processing XPath queries on a Hadoop cluster. In [10], Chen et al. proposed an automatic parallelization approach called pFXQL to describe parallel query plans and proposed a cost model to effectively estimate both computational cost and parallel cost. They implemented our approach in the XQuery engine and conducted experiments on various multi-core systems. PAXQuery [11] and VXQuery [12] are systems for processing small XML documents across a cloud computing cluster, which were not designed to evaluate queries on massive documents. Yet, the problem of XML querying using the MapReduce framework has not received considerable attention, and querying massive XML data remains challenging.

An XML Twig query is essentially a complex selection on the structure of an XML document and can be used to locate element nodes on the data tree corresponding to the XML document. Twig pattern nodes may be elements, attributes and character data. Twig pattern edges are either a *Parent-Child* (*P-C*) relationship (denoted by “/”) or *Ancestor-Descendant* (*A-D*) relationship (denoted by “//”).

For example, an XML document in Fig. 1, indicating that all information *book*, or a given document tree in Fig. 2, and a Twig query in Fig. 3(a) as tree representation and a Twig query using XPath query language, can equivalently be expressed as:

```
book[title = 'XML']//subsection[. = 'twig']
```

The matching results satisfy the following two conditions:

- (1) Matching each binary structural relationship against XML documents;
- (2) “Stitching” together these basic matches.

For example, Fig. 3(b) shows the structural relationships of all the subquery patterns corresponding to the query tree pattern in Fig. 3(a). Given a query twig pattern Q and an XML document D , a match of Q in D is identified by a mapping from the nodes in Q to the nodes in D , so that [5]:

- (1) Query node predicates are satisfied by the corresponding document nodes (images under the mapping);
- (2) The structural (*P-C* and *A-D*) relationships between query nodes are satisfied by the corresponding document node.

The first holistic Twig pattern query join algorithm is TwigStack [5], a basic algorithm for XML documents which utilizes the *L-Stream* representation. It is the core XML data query processing operation, the performance of which directly affects the processing efficiency of the corresponding XML data query. Matching a Twig query means finding all the instances of the query tree embedded in the XML data tree.

There are two key issues related to matching:

- (1) How to quickly determine the structural relationship between two nodes in the XML document tree, including the *P-C* relationship (denoted by “/”) and the *A-D* relationship (denoted by “//”);
- (2) How to efficiently identify the Twig patterns of structural relationships for all of the data in a given XML document.

Traditional Twig query algorithms can be divided into two categories: single-phase algorithms and two-phase algorithms.

Single-phase algorithms find all of the results that match a Twig query in one step; examples of such algorithms include Twig²Stack [13] and TwigMix [14]. Twig²Stack uses a region-encoding labeling scheme and requires the buffer of the entire document in the worst case.

Two-phase algorithms generate some (but not all) solutions of individual query root-to-leaf paths in the first step, and the obtained solutions are merge-joined to compute the answers to the Twig query in the second step. Examples of such algorithms include TwigStackList [15], TJFast [16], and PPSTwigStack [17]. TwigStack [5] computes answers to a Twig query for the case in which the streams contain nodes from a single XML document. To answer the query, first it finds the query answer using combinations of all query nodes, and then performs an appropriate projection on the returned nodes. In [18], Wojnar et al. proposed an approach that utilizes a verified strategy for structural similarity evaluation. It is able to cope with the fact that DTDs involve several types of nodes and can form general graphs. Li et al. [14] focused on the performance of fuzzy XML, and proposed an efficient one-phase holistic Twig join algorithm. It requires more structure-related information based on the extended region scheme. TwigStackList [15] is I/O optimal for queries with only A-D relationships in branching edges, and can identify a larger class of queries which makes it more optimal than TwigStack. TJFast [16] only scans elements for query leaf nodes and needs to access labels of leaf nodes to find the answer. The key issue in TJFast is to determine whether a path solution can contribute to solutions for the entire twig. Lee et al. [19] proposed TJFast-BNS as the data-access optimization of TJFast. TJFast-BNS identified irrelevant elements by the E2Dewey labeling scheme. In [20], Zuo et al. proposed a novel and complete parallel optimization framework for multi-threaded processing of queries on XML documents, based on the query partitioning approach for CMP systems. In [21], Shnaiderman et al. proposed parallel Twigstack algorithms for matching XML query Twig patterns on parallel multi-threaded computing platforms, using the *L-Stream* representation scheme. Staworko et al. [22] studied the question for a given class of queries, and showed that while the union of twig queries is not characterizable, twigs alone were but may require an exponential numbers of examples. Liu et al. [23] presented a labeling scheme to support queries for both static and dynamic possibilistic XML documents. Different from the prior work, they adopted a dynamic encoding scheme and dynamic possibilistic XML documents. Bai et al. [24] presented a specific tuple to represent spatiotemporal data in XML. They also studied the problems of querying fuzzy spatiotemporal data in XML by adding temporal and spatial attributes [25]. In [26], Ma et al. proposed a fuzzy labeling scheme to capture the structural information of fuzzy documents and compute the membership information in order to process the twig queries. Based on this, they devised a twigs algorithm by extending the Dewey code to mark spatiotemporal data. Ma et al. [26] proposed an algorithm for matching a twig pattern query with the AND/OR-logic in fuzzy XML based on the encoding scheme.

In a distributed environment, each machine stores only several fragments of an the XML document. Twig queries must ensure the completeness and correctness of the query results, and minimize the cost of communication among individual machines. The centralized Twig algorithm processes XML document fragments related to other machine nodes, and then re-sends the query, which increases the cost of communication.

To utilize the advantage of a distributed environment, and to reduce the cost of computation and communication, the authors collected the XML document fragments across all machines and sent them to the coordinator site, which is effective for processing XML queries in a de-centralized system in [27]. In [28], Cong et al. proposed a distributed XML query processing algorithm using a partial evaluation strategy. In [9], Choi et al. designed and presented a HadoopXML system to solve the problem of massive multi-twig XML data queries based on Hadoop. Machdi et al. proposed general parallel methods for holistic Twig join algorithms to query XML documents on multi-core systems [1,29]. The proposed parallel methods are based on the data partitioning approach and on the idea of task parallelism (i.e., decomposing the main algorithm into several main tasks, which are then pipelined to create parallelism). In [30], Bi et al. proposed a 3-phase distributed algorithm called DisT3 based on a node distribution mechanism to avoid unnecessary intermediate results. After this, they proposed an improved 2-phase distributed algorithm called DisT2ReP based on a local index called ReP to further reduce the communication cost. Current research on Twig query processing algorithms that use cloud computing are at a very preliminary stage, and related academic results are scarce. Lee et al. [31] proposed data structures called an XP-table and algorithms for minimizing the run-time workload of multiple continuous XPath queries over XML streams. Hao. et al. [32] proposed some algorithms for Xpath queries on a large XML tree in parallel and implemented them on a cluster. Damigos et al. [33] proposed a technique for a large amount of XML data and use the Map-Reduce framework to query the integrated data. Each XML document obtained from the sources is transformed properly in order to fit into a predefined, virtual XML structure. Son et al. [34] proposed a column-store method called SSFile for Hadoop-based distributed systems. SSFile increases the actual amount of data processed per task and supports representative columnar execution techniques for query processing. In [35], Hricov et al. evaluated XPath queries over XML documents using the SparkSQL framework. Additionally, they implemented the initial statements concerning the querying process using the Spark SQL system. Twig query processing of XML documents can be done quite efficiently in centralized environments, however the evaluation of massive XML data (on the scale of terabytes and petabytes) remains challenging.

Additionally, many indexing approaches for XML documents have been proposed to support the evaluation of queries. Some research focused on the node indexing approach [36–38], which indexes XML elements and values. However, they must look up all the elements and values to conduct the evaluation. Recently, Ning et al. [39] proposed a NIndex structure to support effective filtering based on complex XPath expressions. In [40], Hsu et al. proposed a CIS-X index scheme, which combines the advantages of the structural summary and query processing methods. Gopinathan et al. [41] proposed a query processing approach using an indexing mechanism, which combined the path into an index. Qadah et al. [42] introduced

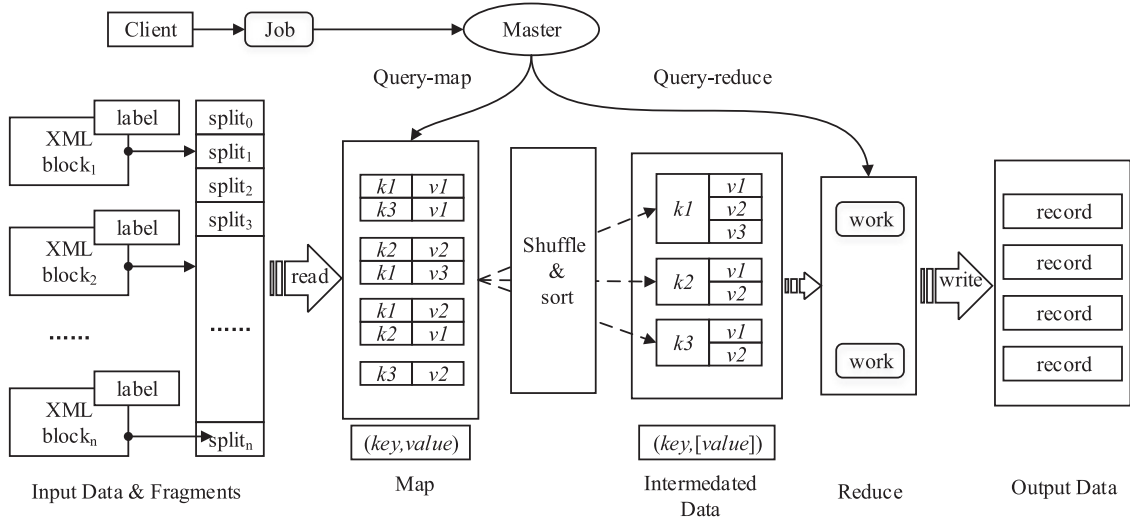


Fig. 4. Workflow of the distributed XML query using the MapReduce framework.

indexing techniques for processing generalized XML documents, including two new algorithms and the associated index structures in processing inter-linked XML documents.

2.3. HDFS and MapReduce framework

The Hadoop distributed file system (HDFS) is a Java-based file system that provides scalable and reliable data storage. HDFS provides high-throughput access to application data and is suitable for applications that use large datasets.⁸ Files stored in HDFS are automatically divided into blocks (which are typically 16–128 Mb for most HDFSs), replicated, and distributed to the nodes' local disks. HDFS exposes a file system namespace and allows user data to be stored. It shares many similarities with existing distributed file systems.

However, the differences between HDFS and other distributed file systems are significant. HDFS exposes a file system namespace and allows user data to be stored in files.⁸ Internally, a file is split into one or more blocks and these blocks are stored in a set of *DataNodes*. The *NameNode* executes file system namespace operations such as opening, closing, and renaming files and directories. The *DataNodes* are responsible for serving read and write requests from the file system clients. The *DataNodes* also perform block creation, deletion, and replication upon instruction from the *NameNode*.

MapReduce [2] is the heart of Hadoop, inspired by functional programming for distributed data processing, and is widely acknowledged by both academia and industry as an effective programming model for querying large-scale data. MapReduce expresses computations using two operators (*Map* and *Reduce*), schedules their execution in parallel on dataset partitions, and guarantees fault tolerance through replication. The *Map* operation processes dataset partitions in a completely parallel manner. The output of the *Map* phase is in the form of $\langle \text{key}, \text{value} \rangle$ pairs. The *Reduce* operation then processes and sorts the $\langle \text{key}, \text{value} \rangle$ pairs received from the *Map* phase, and aggregates all the results. A MapReduce job usually splits the input dataset into independent chunks that are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then provided to the reduce tasks. Typically, both the input and the output of a job are stored in a file-system. A MapReduce distributed system features a Master node and Workers. The master node handles data partitioning and schedules tasks automatically on an arbitrary number of workers. Once the functions are specified, the runtime environment automatically schedules the execution of *Mappers* on idle nodes. Each node executes a *Map* function against its local dataset partition, writes intermediate results to its local disk and periodically notifies the master about its progress. As the *Mappers* produce intermediate results, the master node assigns *Reduce* tasks to idle nodes.

3. Parallel XML queries

When processing a distributed XML query on a large amount of XML data, we use the HDFS as the data storage system and MapReduce as the programming framework. In this section, we present the process, which involves three steps, as shown in Fig. 4.

First, we split up the XML data using *XMLInputFormat*, which decides how to partition an input file into *file-splits*, suitable for XML grammar rules. Then the *file-splits* are provided as input to the distributed storage system. Normally, these parsing

⁸ http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

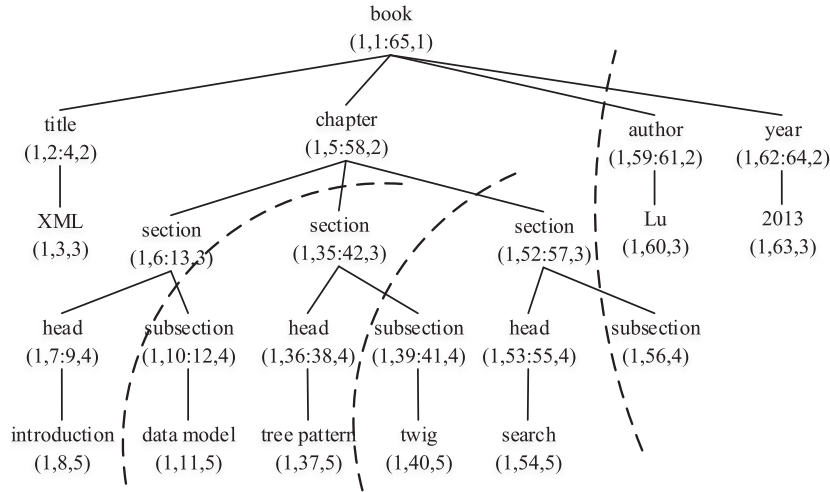


Fig. 5. Partitioning a document into four fragments by type.

and partitioning operations are only executed once for a massive XML document. In the second and third steps, we execute a parallel query, which includes a distributed XPath query and a Twig query. The results of these queries are presented to the MapReduce job on the Hadoop platform. During the *Map* operation, clusters perform parallel computations for XML queries on all computers that store XML data splits, while in the *Reduce* operation, all the intermediate results are collected to generate the final result.

3.1. Splitting XML data with *InputFormat*

As previously mentioned, an XML file cannot be split because XML features opening and closing tags. This means that processing cannot be initiated at an arbitrary point between these bounding tags.

To address this issue, we split the XML data using *InputFormat*, which decides how to split an input file into *file-splits* for a distributed storage system. The basic *InputFormat* interprets each line in a *file-split* as a *key-value* pair where the *key* is a byte offset address in the input file and the *value* is the context of the line. Thus, proper splitting is determined not only by the size of the file, but also by logical constraints on XML data.

For example, suppose we have an XML document as the one shown in Fig. 1. This document includes the text “<author>Lu</author>”. The document is split into two *file-splits* where f_{si} represents “<author>Lu</” and f_{si+1} represents “author>”. This split introduces a fatal problem related to XML grammar because the end-tags of all elements must be in the form of </tagname>.

Two approaches to correctly split an XML document are presented in this paper:

1. Partitioning the document into fragments by type.
2. Splitting the XML data using *virtual nodes*;

Each of these two approaches has its own advantages and is useful for specific query types. For example, the first approach can solve parallel Twig queries efficiently, while the second approach is effective in solving parallel XPath queries.

3.1.1. Partitioning the document into fragments by type

To avoid fatal errors related to XML grammar that may arise during document splitting, we need to ensure that the input format conforms to XML grammar requirements.

During the loading phase, the input XML document is parsed and stored according to the XML storage scheme. We present a simple API for XML (SAX)⁹ for parsing. The SAX parser parses the analyzed XML file line by line and triggers events when it encounters an opening tag, a closing tag, or character data in the XML file.

During partitioning, the boundaries of all *file-splits* are adjusted so that no tags are separated in adjacent *file-splits*. The approach is to check the boundary of each *file-split* for possible tag separations, and extend the *file-split* boundaries to include tags. Furthermore, the records in a *file-split* are also interpreted on the basis of tags, and each record should contain a tag and a value, as shown in Fig. 5.

As shown in Fig. 6, massive XML data files can be divided into several partitions. To retain connections between different fragments, marks should be introduced. Obviously, any node can be divided into one of the following three types:

⁹ <http://www.saxproject.org/>.

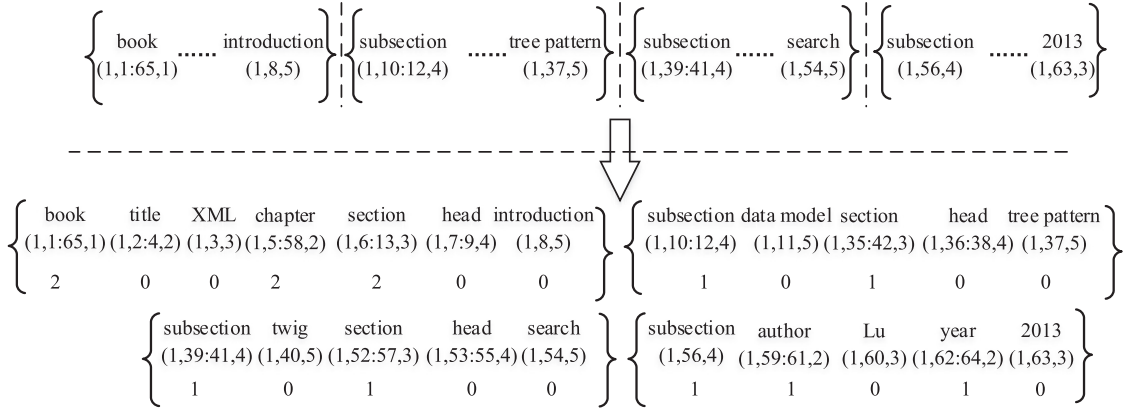


Fig. 6. Partitioning a document into four fragments with types.

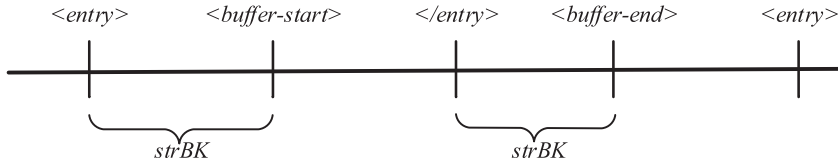


Fig. 7. Example of finding a close tag in a buffer.

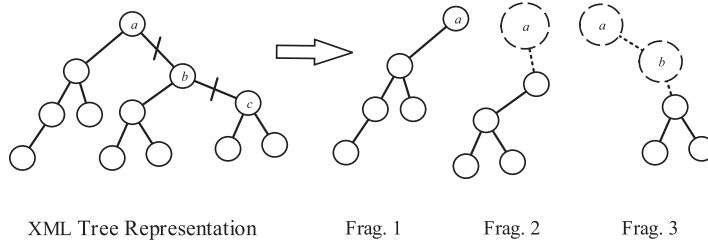


Fig. 8. Partitioning the XML tree into three fragments using virtual nodes.

- (1) Type 0: neither the parent edge nor the child edge are cut, i.e., the node is intact;
- (2) Type 1: the parent edge is cut, but the child edges are intact;
- (3) Type 2: at least one child edge is cut, but the parent edge is intact;

For the example given in Fig. 7, if a tag `<entry>` is found from the current buffer stream, we mark its position. Sometimes one buffer will not be able to store the entire `entry`; thus we use a dedicated variable `strBK` to memorize the information truncated by `buffer`. After this, `strBK` with the current buffer pieces the whole `<entry>` together. Each node is assigned its own type marker that encodes the effect of partitioning on that node.

For example, the node `<book>` (1, 1:65, 1) is classified as type 2, because at least one child edge is cut, but the parent edge is intact. For all the nodes, their types can be set after parsing using SAX. The node type is partition-dependent, which means it needs to be checked and reset for every partition. The XML document tree can be partitioned into four fragments, as shown in Fig. 6.

We propose XML partitioning Algorithm 1 which decides how to split an input file into file-splits for a distributed storage system in order to split XML data in a way which is suitable for XML grammar. The purpose of `ADDTOPARTITION(S_d)` is to determine the tagging of nodes in a partition.

The purpose of Algorithm 2 is to initiate the XML buffer stream and to check whether data are suitable for XML formatting. `BUFFERSTREAMINITIATE(X_d)` sets `StartPos`, `EndPos`, and node type. Function `CHECKPARTITION()` rechecks the type values of the nodes, to determine whether at least one child edge is cut but the parent edge is intact.

3.1.2. Splitting the XML data using virtual nodes

A massive XML data file should be partitioned as a collection of fragments for parallel processing. To retain connections between the file fragments, another approach is to use *virtual nodes* with each partition. The XML tree in Fig. 8 is split into three fragments using *horizontal fragmentation*. Notice that Frag.2 and Frag.3 are obtained by splitting the XML tree including $\{a\}$ and $\{a, b\}$, respectively.

Algorithm 1 XML Partitioning by Type.**Input:** Initialized XML document X_d .**Output:** Partitions $prtCollections$. //data fragments stored in distributed system

```

1: function ADDTOPARTITION( $S_d$ )
2:   Initialize  $buffer\ b$ ,  $bufferPosn$ ,  $bufferLength$ ; //memorize the information in  $b$ 
3:    $bufferStream\ S_d = bufferStreamInitiate(X_d)$ ; //see Algorithm 2 Function 1
4:   while !eof( $S_d.bufferStream$ ) do
5:     if  $bufferPosn < bufferLength$  then
6:       if  $bufferStream\ node\ N_i == prt.head$  then
7:          $N_i.setType(1)$ ; //set the node into type 1
8:          $prt.push(b)$ ; //push the node into buffer  $b$ 
9:       end if
10:    end if
11:    if  $bufferPosn == bufferLength$  then
12:      if ! $N_i.isleaf()$  then
13:         $N_i.setType(2)$ ; //set the node into type 2
14:         $prt.push(b)$ ; //push the node into buffer  $b$ 
15:      end if
16:    end if
17:     $prts$  add to  $prtCollection$ ; //add the file-split as prts into  $prtCollection$ 
18:     $N_i.CheckPartitionwithTypes()$ ; //see Algorithm 2 Function 2
19:    flush  $b$ ;
20:  end while
21: end function

```

Algorithm 2 XML Buffer Stream Initiate and Check Partition.**Input:** Initialized XML document X_d .

```

1: function BUFFERSTREAMINITIATE( $X_d$ )
2:   SAX ( $X_d$ ); //using SAX API
3:   startElement();
4:   { $node.begin = getNodeBegin()$ ;
5:    $S.push(node)$ ;  $bufferStream\ S_d.add(node)$ ;}
6:   endElement();
7:   { $Node\ node = S.pop()$ ;  $node.end = getNodeEnd()$ ;}
8:   return  $bufferStream\ S_d$ 
9: end function
10: function CHECKPARTITIONWITHTYPES( $X_d$ )
11:   if  $N_i.parent$  in  $prt$  then
12:     if  $N_i.parent.type == 0$  then  $N_i.parent.type.setType(2)$ ; //reset the node type
13:   end if
14: end if
15: end function
16: function CHECKPARTITIONWITHVIRTUALNODES( )
17:   Parse  $prt$  conform to XML format
18: end function

```

We propose XML partitioning [Algorithm 3](#) to split XML data using virtual nodes; the algorithm splits the data file into *file-splits* to a distributed storage system. The algorithm performs three functions. Function ADDTOPARTITION(S_d) places a node into a partition. All small fragments add virtual nodes. BUFFERSTREAMINITIATE(X_d) from [Algorithm 2](#) initiates document X_d using SAX and returns *bufferStream* as input to function ADDTOPARTITION(S_d). In addition to these two functions, function CHECKPARTITION() rechecks *prt* for conformance to the XML format.

After partitioning, massive XML files in the Hadoop system are distributed based on the block size of *DataNodes*. While processing using MapReduce, care must be taken in determining the record boundaries, to ensure that all XML records are processed properly, even if a record is split across multiple HDFS blocks.

These two partitioning algorithms are useful in splitting massive XML files into smaller files, and each method is advantageous for specific query types. The second approach handles parallel XPath queries efficiently, while the first approach handles Twig queries efficiently.

Algorithm 3 Partitioning Using Virtual Nodes.**Input:** XML document X_d .**Output:** Partitions $prtCollection$.

```

1: function ADDTOPARTITION( $S_d$ )
2:   Initialize  $buffer\ b$ ,  $bufferPosn$ ,  $bufferLength$ ,  $prt$ ;
3:    $bufferStream\ S_d = bufferStreamInitiate(X_d)$ ; /also see Algorithm 2 Function 1
4:   while !eof( $S_d.bufferStream$ ) do
5:      $prt.addVirtualRoot$ ; //add the virtual root to the current node
6:     for each element after the Virtual Root do
7:       if  $bufferPosn < bufferLength$  then
8:          $prt.push(b)$ ;
9:       end if
10:    end for
11:    flush  $b$ ;
12:     $prt.CheckPartitionWithVirtualNodes()$ ; //see Algorithm 2 Function 3
13:     $prts$  add to  $prtCollection$ ; //add data pieces as  $prts$  into  $prtCollection$ 
14:  end while
15:  return  $prtCollection$ 
16: end function

```

3.2. An overview of parallel XML querying

In this section, we describe the parallel query method, including the process of the distributed XPath query and a Twig query algorithm. The algorithm description is given in Algorithm 4. The results of these queries were presented to the

Algorithm 4 Distributed Query Algorithm.**Input:** query Q , Partitions $prtCollection$.**Output:** $\langle key, value \rangle$.

```

1: function MAP( $\langle key, value \rangle$ )
2:    $q = DistributedCache.get(Q)$ ; //transform  $Q$  into  $q$  identified by Hadoop
3:    $filesplit = context.getInputSplit(prtCollection)$ ; //also transform  $prtCollections$  identified by HDFS
4:   Call XPath-MR or TwigStack-MR;
5: end function
6: XPath-MR( $q.filesplit$ ); //also see Algorithm 5 Function
7: TwigStack-MR( $q.filesplit$ ); //also see Algorithm 6 Function
8: function REDUCE( $key$ ,  $value$ ,  $context$ )
9:   while  $val$  in  $values$  do
10:     $sum.add(val.get())$ ;
11:  end while
12:   $result.set(sum)$ ; //aggregate all results
13:   $context.write(key, result)$ ;
14: end function

```

Hadoop platform. During the *Map* operation, clusters perform parallel computations for XML queries on all computers that store XML data splits, while in the *Reduce* operation all the intermediate results are collected to generate the final result.

3.2.1. Parallel XPath querying

Our XPath-MR algorithm is based on the classical XPath algorithm. The rationale for parallelizing XPath queries is to enable it to process different subtrees of the document tree in parallel. The algorithm is applied in parallel to the segments that were obtained after partitioning the XML file into *file-splits*. Each base XPath algorithm is run on different *ppts* with the XML format.

As shown in Algorithm 4, the distributed XPath-MR algorithm performs two functions. Function XPATHQUERY in Algorithm 5 is the core function of the algorithms, which executes the XPath query over the subtree, aka *fst*, constructed from a *file-split*. The algorithm is very similar to the original XPath algorithm, but is extended to use in the MapReduce framework. Function REDUCE() collects all XPath query results over the subtrees.

3.2.2. Parallel Twig-MR algorithm

Our parallel TwigStack-MR algorithm is another XML basic query, which is based on the classical Twig algorithm. The basic idea for parallelizing twig pattern queries is to process different subtrees of the document tree. The algorithm de-

Algorithm 5 Main Body of XPath-MR Algorithm.**Input:** XPath query q , File split $filesplit$.

```

1: function XPATHQUERY( $(q, filesplit)$ )
2:    $fst=filesplit.ConstructTree()$ ;
3:   while !eof( $fst$ ) do
4:      $node=xmlParse.selectNode(fst,q)$ ;
5:      $value=node.toString();context.write(key, value)$ ;
6:   end while
7: end function

```

scription is given in Algorithm 4. The results of these queries were presented to the Hadoop platform. In the parallel run, the algorithm is run on each subtree separately based on the method of partitioning the XML document tree into *file-splits* which contains four fragments with types, so that each base TwigStack algorithm can run on a different subtree. Function TWIGSTACK-MR in Algorithm 6 executes the twig query over the subtree. It is very similar to the original TwigStack

Algorithm 6 Main Body of TwigStack-MR Algorithm.**Input:** Twig query q , $filesplit$.

```

1: function TWIGSTACK-MR( $q, filesplit$ )
2:    $fst = filesplit.ConstructTree()$ ; //construct tree based on filesplit
3:   while  $q.LevelNum < fst.LevelNum$  do
4:      $q = q.getFirstChild()$ ;
5:   end while
6:   while !eof( $q$ ) do
7:      $GetNext(prt.Collection, q_{act})$ ; //see Algorithm 7
8:      $cleanStacks()$ ;
9:     if  $isRoot(q_{act}) \parallel !empty(Sparent(q_{act}))$  then
10:       $cleanStacks(Sq_{act}, next(q_{act}))$ ;
11:       $moveStreamToStack(Tq_{act}, Sq_{act}, pointerToTop(Sparent(q_{act})))$ ;
12:    end if
13:    if  $q_{act}.isLeafNode$  then
14:       $showSolutions(Sq_{act})$ ; //see Algorithm in Ref. [6]
15:    else
16:       $advance(Tq_{act})$  //also see Algorithm in Ref. [6]
17:    end if
18:  end while
19:   $mergeAllPathSolutions(result)$ ;
20:   $value=getResult()$ ;
21:   $context.write(key, value)$ ;
22: end function

```

algorithm [5], but is adjusted to the MapReduce framework.

The algorithm proceeds in two steps. In the first step (lines 13–17), some (but not all) solutions to individual query *root-to-leaf* paths are computed. In the second step (line 19), these solutions are merge-joined to compute the answers to the query twig pattern. Function REDUCE collects all twig query results over the subtrees.

$GetNext(q)$ (Algorithm 7), presented in [5] for the first time, recursively invokes function GETNEXT for each q_i in children(q). If none of the returned nodes q_i is equal to n_i , the algorithm immediately returns q_i . Otherwise, the algorithm tries to locate a child of n that satisfies the above three properties.

3.2.3. Distributed XPath querying using partition index

Furthermore, we use function MAP() to construct the index, as shown in Algorithm 8. If *unique identifier* is used as the index keyword, the corresponding entry can easily be found. When using MapReduce on HDFS, each row of data has its corresponding file offset, and reading can start from that offset to make the process more efficient. The *offset* is set as the value of the index, as shown in Fig. 9.

The XPath index can be built according to the document's characteristics. Because offset registers the start position of the entry element, we can use it as *key* in function MAP(). If the XML document contains the entry id, the function context.write saves the id and *offset* as a *key* and *value*. Function MAP() delivers the output $\langle key, value \rangle$ to shuffle before function REDUCE().

During this process, values with the same *key* are grouped together, and these values are then placed in an iterator interface for function REDUCE(). Because one *key* has one *value*, the Reduce function only needs to take the first value from

Algorithm 7 GetNext(q) Algorithm.

```

1: function GETNEXT( $q$ )
2:   if isLeaf( $q$ ) then
3:     return  $q$ ;
4:   end if
5:   while  $q_i$  in children( $q$ ) do
6:      $n_i$  = GetNext( $n_i$ )
7:     if  $n_i \neq q_i$  then
8:       return  $n_i$ 
9:     end if
10:  end while
11:   $n_{min}$  = minarg(nextBegin( $T_{n_i}$ ));
12:   $n_{max}$  = maxarg(nextBegin( $T_{n_i}$ ));
13:  while nextEnd( $T_q$ ) < nextBegin( $T_{n_{max}}$ ) do
14:    advance( $q$ )
15:  end while
16:  if nextBegin( $T_q$ ) < nextBegin( $T_{n_{max}}$ ) then
17:    return  $q$ 
18:  else
19:    return  $n_{min}$ 
20:  end if
21: end function

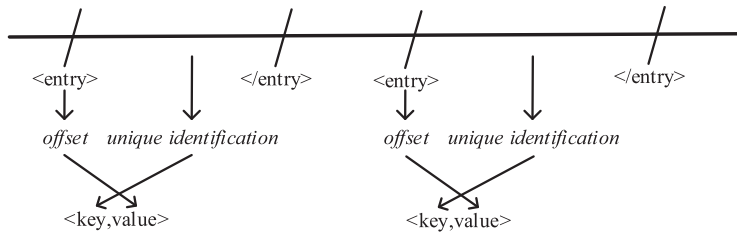
```

Algorithm 8 Distributed XPath Index Construction.**Input:** XML document X_d .**Output:** $\langle \text{key}, \text{value} \rangle$.

```

1: function MAP( $\text{key}, \text{value}, \text{Text}$ )
2:   if Text.contains(entry) then
3:     offset = key;
4:   end if
5:   if Text.contains(entry id) then
6:     context.write(new Text(id), offset);
7:   end if
8: end function
9: function REDUCE( $\text{key}, \text{value}, \text{context}$ )
10:  iter = values.iterator();
11:  while iter.Next is not null do
12:    offset.set(iter.Next.toString());
13:    context.write(key, offset);
14:  end while
15: end function

```

**Fig. 9.** Distributed XPath index construction.

$iter$ objects, and then write the key to the output file. When the size of the index file is larger than that of the fragment set, the index file should be divided into several partition indices, with each partition index corresponding to a map task query. This can improve the query efficiency, as shown in Fig. 10.

To process large index files, we design the query using MapReduce and query block in parallel. As shown in Algorithm 9, the keyword in function MAP() is given an id attribute that is determined by the index construction. The value of the at-

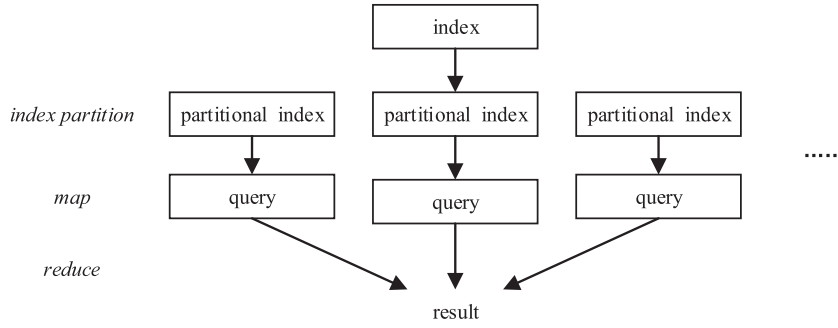


Fig. 10. Distributed XPath querying using partition index.

Algorithm 9 Distributed XPath Querying using Index.

Input: XPath query Q , Index Partitions $IdxPrtCollection$.

Output: $\langle key, value \rangle$.

```

1: function MAP( $key, value, Text$ )
2:   if Text.contains(keyword) then
3:     context.wirte(new Text( $key$ ), new Text( $value$ ));
4:   end if
5: end function
6: function REDUCE( $key, value, context$ )
7:   while val in values do //whether the keyword appears in the index file
8:     sum.add(val.get());
9:   end while
10:  result.set(sum); //aggregate all results
11:  context.write( $key, result$ );
12: end function
  
```

tribute id is determined based on whether this keyword appears in the index file. Function REDUCE() collects all the results over the subtrees.

HDFS provides an interface to random access files, as shown in Algorithm 10. Hence, we used function CopyBytes (in) to

Algorithm 10 Random Access File Algorithm.

Input: XML document X_d .

Output: $\langle key, value \rangle$.

```

1: function RANDOMACCESSFILE( $filePath, offset$ )
2:  FileSystem fs = getFileSystem(filePath); //Initialize fs
3:  FSDataInputStream in = fs.Open(filePath); //Initialize data input stream
4:  in.Seek(offset); //see data based on offset
5:  OneEntry = CopyBytes(in);
6:  CloseStream(in);
7:  return OneEntry
8: end function
9: function XPATHQUERY( $q, OneEntry$ )
10:  oey= OneEntry.ConstructTree();
11:  while !eof(oey) do
12:    node = xmlParse.selectNode(oey, q)
13:    value= node.toString();
14:  end while
15:  return value
16: end function
  
```

locate the entry and directly read the file based on the offset. In the algorithm, the offset is the number of bytes from the beginning of the file location to the specified location.

Table 2
Statistical Characteristics of XML Datasets.

Filename	File size (KB)	#of elements	Average depth	#of distinct paths
Dblp2016	1,792,032	43,168,099	3.55	156
UniRef100	90,007,328	1,391,576,362	4.32	33
iProClass	176,313,116	4,254,446,438	5.52	248
UniProtKB	259,392,498	4,897,707,134	4.23	271
UniParc	238,483,766	598,532,092	3.61	16

Table 3
XPath queries selection.

Name	QueryTrees
DBQ ₁	/dblp/article[@key = \'journals/acta/Saxena96\']/title
DBQ ₂	//article[@key = \'journals/acta/EngelrietV88\']/title
URQ ₁	/UniRef100/entry[@id = \'Q6GZX3\']/repMember
URQ ₂	//entry[@id = \'Q6GZX4\']/repMember/dbReference[@id = \'FRG3G\']
PCQ ₁	/iProClassDatabase/iProEntry[@ID = \'Q6GZX4\']/SEQUENCE
PCQ ₂	//iProEntry[@ID = \'Q197F8\']/Protein_Name_and_ID/UniProtK

Table 4
Twig queries selection.

Name	QueryTrees
UKQ	uniprot[protein/submittedName]/gene/name
UPQ	uniparc/entry/[accession]/sequence
URQ	UniRef100/entry/[name]/representativeMember/sequence

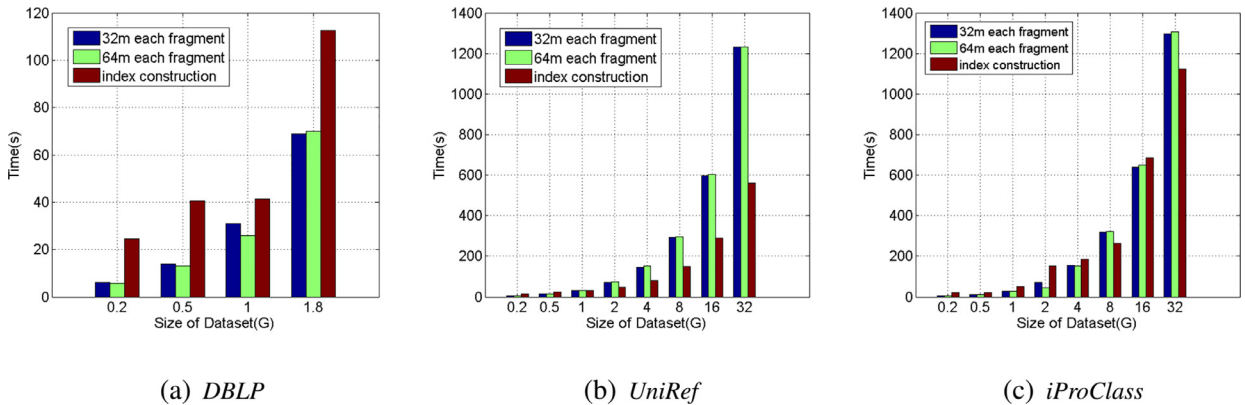


Fig. 11. Time Taken for File Splitting and Index Construction, for Different Datasets.

3.3. Implementation using the MapReduce framework

The Hadoop system divides a computer cluster into a master node and multiple worker nodes. The master node is responsible for task scheduling, resource allocation, and error management. Worker nodes process Map tasks and Reduce tasks in parallel.

Distributed XML querying is implemented in the MapReduce framework, as follows.

1. Data Reading.

Present *fs put()* of Hadoop to load the *file-splits* into a distributed dataset, then use this dataset as the input to *map()* to start Map tasks.

2. Map Tasks.

Because all Map tasks use the same algorithm, a task is constructed. The algorithm's calculation is mainly done in the Map stage. The outputs of *map()* are *key-value* pairs, in which the distance is *key* and the path is *value*.

3. Reduce Tasks.

In the Reduce stage, all of the results are aggregated.

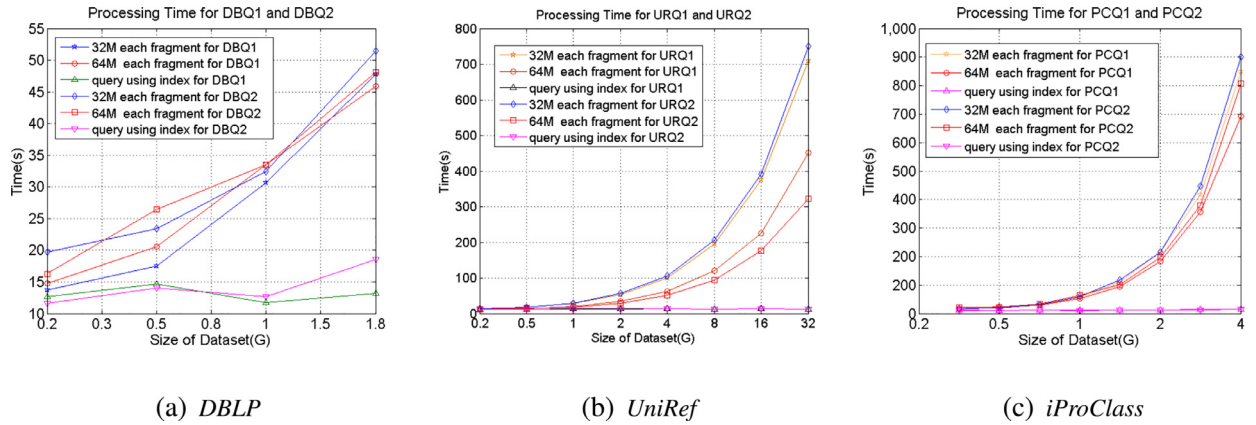


Fig. 12. Query Size Variations Across Different Datasets.

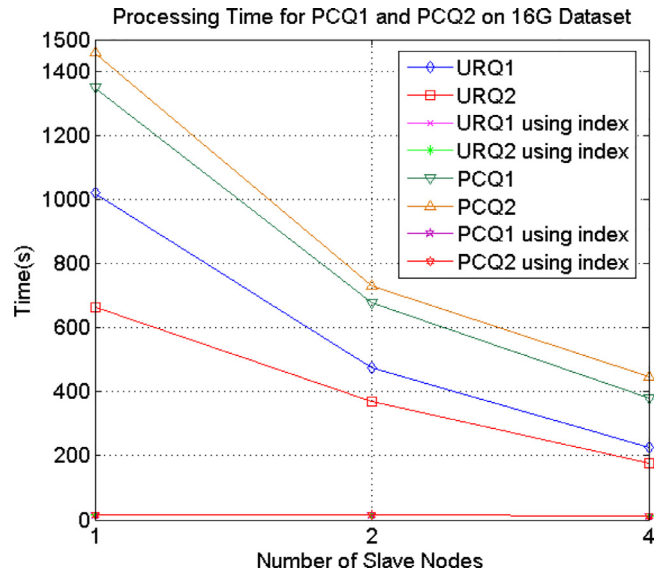


Fig. 13. Correlation between the Processing Time and the Number of Data Nodes.

4. Performance evaluation

4.1. Experimental setup

In the following, we describe the experimental validation of our algorithm in relation to file splitting, construction of the index, and evaluation of the XPath algorithms.

In our experiment, we use real-world XML data. Specifically, we use *DBLP*¹⁰ which is a computer science bibliography website that contains bibliographic entries on more than 3.1 million journal articles, conference papers, and other publications in the field of computer science. Another website, UniProt Reference Cluster database (*UniRef*¹¹), contains data on clustered sets of sequences from the UniProt Knowledgebase (*UniProtKB*) and selected records that enable complete coverage of the sequence space at several resolutions. It combines identical sequences and sub-fragments with 11 or more residues from any organism. Finally, the *iProClass* database¹² provides value-added descriptions of proteins and serves as a framework for data integration in a distributed networking environment, with links to over 160 biological databases.

Table 2 lists the statistical characteristics of the *DBLP*, *UniRef*, *iProClass*, *UniProtKB*, and *UniParc* datasets, including size of file, number of elements in each set, average depth, and number of distinct paths.

¹⁰ <http://dblp.uni-trier.de/>.

¹¹ <http://www.uniprot.org/>.

¹² <http://pir.georgetown.edu/pirwww/dbinfo/iproclass.shtml>.

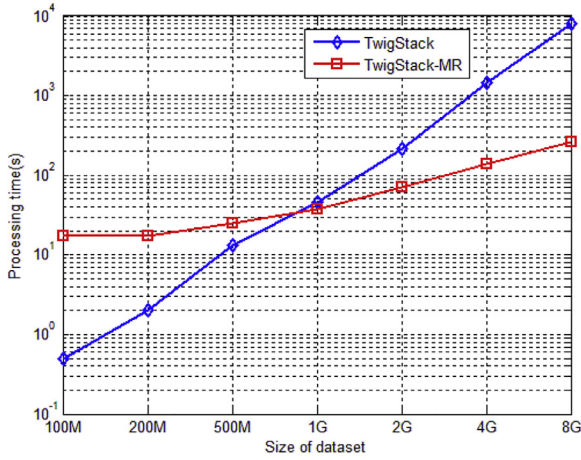
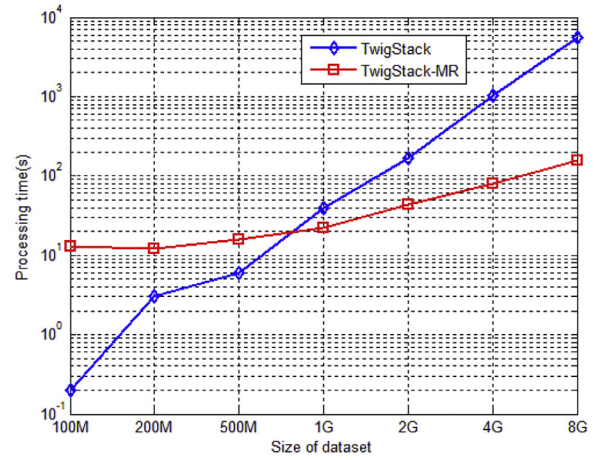
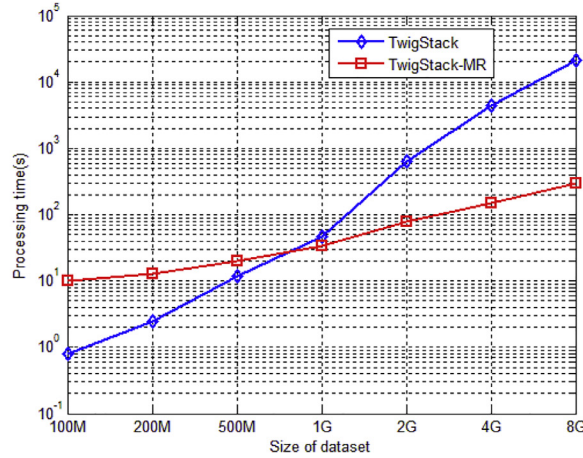
(a) $UKQ(a)$ (b) $UPQ(b)$ (c) $URQ(c)$

Fig. 14. Processing time vs. the dataset size, for TwigStack and TwigStack-MR. (a) UKQ , (b) UPQ , and (c) URQ .

Algorithm 4 is a consistent XML query algorithm for a distributed environment. In order to facilitate an experimental comparison, we describe the experimental validation of our Algorithm 5 and Algorithm 6 separately.

We chose six XPath queries, as listed in Table 3. To facilitate a comparison of different experiments, we use the entire DBLP dataset and subsets of the other datasets, i.e., the maximal volume of the other datasets is 32 G. The query trees are abbreviated as DBQ_1 , DBQ_2 , URQ_1 , URQ_2 , PCQ_1 and PCQ_2 , while DBQ , URQ and PCQ are the XPath queries for DBLP, UniRef100, and iProClass dataset, respectively.

We also evaluate our Twig query algorithm on real-world data to evaluate the TwigStack-MR algorithms. Table 2 lists the statistical characteristics of the three real-world sets of XML data that we used in our experiments.

We chose three twig pattern queries as shown in Table 4. The twig pattern queries are for different datasets. To facilitate a comparison across different experiments, we use a subset of these datasets, i.e., the maximal volume of each tested dataset was 64 G. The query trees are abbreviated as UKQ , UPQ , and QRQ . UKQ is the twig query for UniProtKB, while UPQ and URQ are the twig queries for UniParc and UniRef100 dataset, respectively.

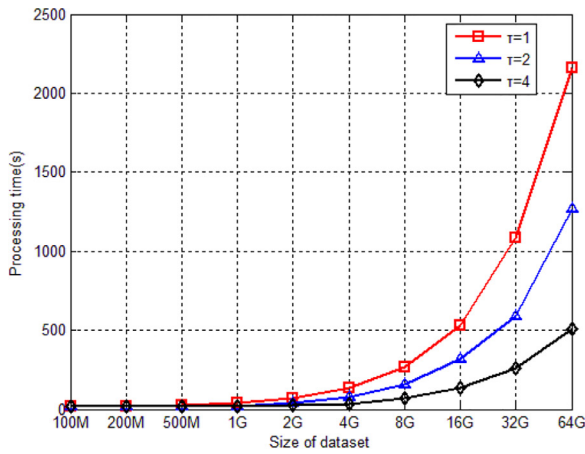
All these experiments are performed on a cluster of 4 machines that ran Ubuntu 14.04 LTS. Each node is equipped with an Intel Xeon E5-2609 2.5GHz CPU and 32 GB of memory. We use Hadoop as an open source implementation of the MapReduce framework written in Java. The version of Hadoop used is 2.7.1.

4.2. Evaluation of the parallel XPath query method

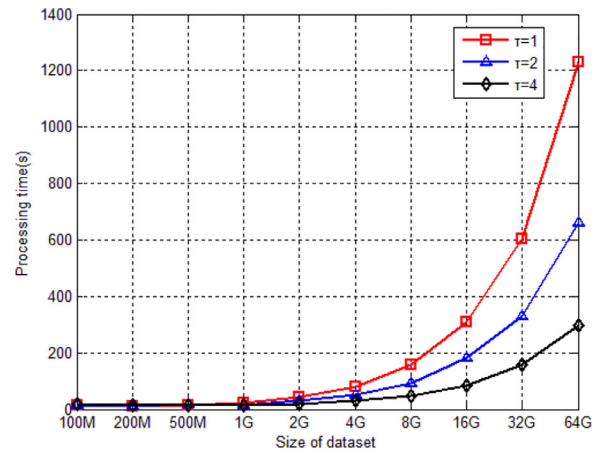
We design three groups of experiments to test the efficiency of the file-splitting and index-construction processes, as shown in Fig. 11. This shows the time required for file splitting and index construction, vs. the size of dataset, for dataset sizes ranging from 0.2 G to 1.8 G (for the *DBLP* database), and for dataset sizes ranging from 0.2 G to 32 G (for the *UniRef* and *iProClass* databases). The results suggest that the time needed to split data files into sets of fragments with virtual nodes is about the same for all datasets. Yet compared with the time for file splitting, the time for index construction is longer for the *DBLP* database and nearly double for the *UniRef* database, because the datasets are larger in size.

The results for the XPath queries in Fig. 12 were obtained with/without index processing. These results indicate that the efficiency also varies as the dataset size varies from 0.2 G to 1.8 G (for *DBLP*) and from 0.2 G to 32 G (for *UniRef* and *iProClass*, respectively). These results show that for small datasets, the performance time of the proposed distributed XPath query method is nearly the same, regardless of indexing. The time increases significantly for both 32M and 64M fragments. But index-based querying becomes advantageous for large XML files, especially above 2 G.

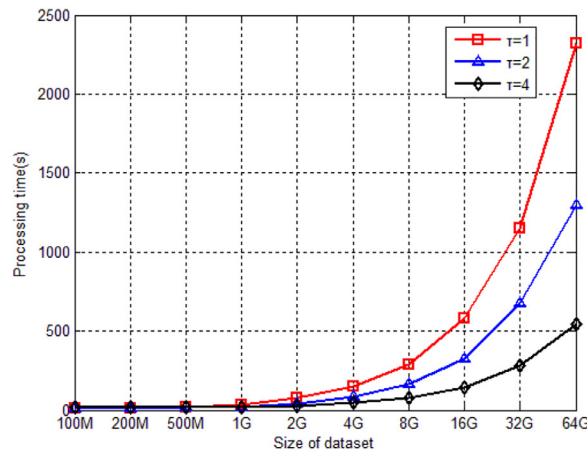
In addition, the time needed for index-based querying is very stable, about 12s, with slight fluctuations across datasets of different sizes. Although the query time accounts for all events of random access to HDFS, the access time is under 0.3s, as shown in Fig. 12(b) and 12(c). Thus, the access time can be ignored compared with the entire query time.



(a)



(b)



(c)

Fig. 15. Correlation between processing time and dataset size for (a) UKQ, (b) UPQ, and (c) URQ.

4.2.1. Scalability of the XPath query method

We evaluate the scalability of the proposed architecture by measuring the correlation between execution time and number of Hadoop cluster nodes. The DBLP dataset is relatively small; thus, we used a subset of UniProt and iProClass datasets (16 G). As the number of nodes increased from 1 to 4 (4X), the execution time increased from 3.2X to 3.8X, as shown in Fig. 13. This is because starting up the MapReduce framework consumed the majority of the execution time.

The execution time is quite short, especially for index-based queries, despite the fact that the system is tested on a relatively small cluster. It is worth noting that the processing time decreased monotonously with an increase of the node number, which indicates good scalability.

4.3. Evaluation of the parallel Twig query method

We design four groups of experiments to test the efficiency and scalability of the twig pattern querying method. The efficiency results are compared in Fig. 14(a), (b), and (c).

4.3.1. Efficiency

Fig. 14(a), (b), (c) shows variations in efficiency as the dataset size increases from 0.1 G to 8 G. These figures indicate that the traditional TwigStack method first performs better in speedup, because starting in MapReduce consumes a major chunk of the query time for relatively small datasets. However, the traditional TwigStack method becomes less advantageous for large XML files, especially above 1 G. The TwigStack-MR method gains significantly in sizeup when the scale of XML data increases.

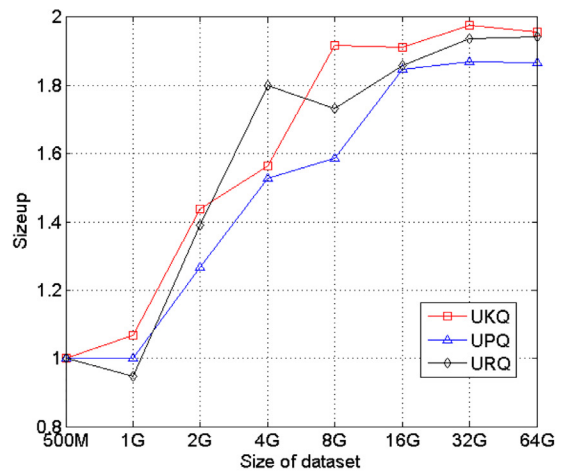
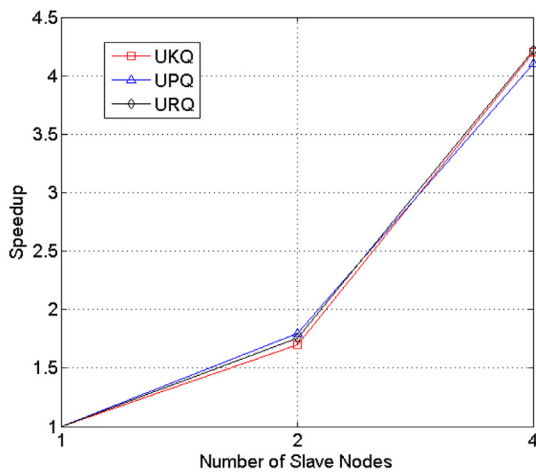
We evaluate the scalability of the proposed architecture by measuring the execution time as the number of Hadoop cluster nodes increases.

All these experiments are performed on a small cluster. The number of nodes in the cluster varies from 1 to 4. Fig. 15(a), (b), and (c) indicates the correlation between speedup and cluster size up to 64 G. When all four nodes are used at the same time, the execution time of the proposed algorithm is about 9 min, 5 min, and 8.5 min, for UKQ, UPQ, and URQ, respectively. Hence, the execution time is quite short, despite the fact that we tested the system on a relatively small cluster.

4.3.2. Evaluation of speedup and scaleup

Fig. 16(a) presents the speedup and scaleup comparisons. We evaluate the speedup of our algorithm by varying the number of nodes in the cluster from 1 to 4. As the number of nodes increased from 1 to 4 (4X), the speedup increased by 4.25X, 4.14X and 4.23X for queries on different sets. The system's performance on distributed queries significantly improves as the number of clusters increases.

The correlation between performance and the dataset size is shown in Fig. 16(b). Clearly, significant improvement in performance is observed with the expansion of the scale of the analyzed XML data, because starting up MapReduce consumes a major chunk of the query time for relatively small datasets. It is worth noting that a robust increase is observed in all cases, which suggests good scalability. Hence, distributed twig querying is more advantageous for larger-size XML data.



(a) Performance gain vs. the number of nodes, for UKQ, (b) Performance gain vs. the data size, for UKQ, UPQ, and URQ

Fig. 16. (a) Speedup and (b) Sizeup, for UKQ, UPQ, and URQ.

5. Conclusions

Motivated by the lack of support for the efficient processing of large-scale XML queries on Big XML data, including parallel XPath querying and parallel Twig querying, we propose a MapReduce-based approach to process large volumes of XML data. We split a large-scale XML data file into file-splits to be served as inputs to a distributed storage system. These file-splits are processed by the distributed query algorithm, by which different fragments of the document tree are processed in parallel. The algorithm can also be used in distributed cloud environments based on the MapReduce framework for efficient querying. Furthermore, we present a partition index for the efficient identification of node entries. The experiments show that our approach is efficient and sufficiently scalable.

In the future, we plan to study the labeling XML data in parallel, and use other node labeling schemes to improve the system's performance.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the [National Natural Science Foundation of China](#) under grant no. [61672046](#), and La Trobe University 2017 CSRC Staff Exchange Program for Collaborative Research.

References

- [1] I. Machdi, T. Amagasa, H. Kitagawa, Parallel holistic twig joins on a multi-core system, *Int. J. Web Inf. Syst.* 6 (2) (2010) 149–177.
- [2] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, *Commun. ACM* 51 (1) (2008) 107–113.
- [3] H. Fan, D. Wang, J. Liu, Distributed XPath query processing over large XML data based on MapReduce framework, in: 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery, ICNC-FSKD 2016, Changsha, China, August 13–15, 2016, 2016, pp. 1447–1453.
- [4] H. Fan, H. Yang, Z. Ma, J. Liu, TwigStack-MR: an approach to distributed XML twig query using MapReduce, in: 2016 IEEE International Congress on Big Data, San Francisco, CA, USA, June 27–July 2, 2016, 2016, pp. 133–140.
- [5] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002, pp. 310–321.
- [6] L. Liu, M.T. Özsu (Eds.), *Encyclopedia of Database Systems*, Springer, US, 2009.
- [7] N. Bidoit, D. Colazzo, N. Malla, C. Sartiani, Partitioning XML documents for iterative queries, in: Proceedings of the 16th International Database Engineering & Applications Symposium, 2012, pp. 51–60.
- [8] G. Cong, W. Fan, A. Kementsietsidis, J. Li, X. Liu, Partial evaluation for distributed XPath query processing and beyond, *ACM Trans. Database Syst.* 37 (4) (2012) 32:1–32:43.
- [9] H. Choi, K. Lee, S. Kim, Y. Lee, B. Moon, HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, 2012, pp. 2737–2739.
- [10] R. Chen, H. Liao, Z. Wang, H. Su, Automatic parallelization of XQuery programs on multi-core systems, *J. Supercomput.* 72 (4) (2016) 1517–1548.
- [11] J. Camacho-Rodríguez, D. Colazzo, I. Manolescu, PAXQuery: efficient parallel processing of complex XQuery, *IEEE Trans. Knowledge Data Eng.* 27 (7) (2015) 1977–1991.
- [12] E.P.C. Jr., T. Westmann, V.R. Borkar, M.J. Carey, V.J. Tsotras, Apache VXQuery: a scalable XQuery implementation, *CoRR abs/1504.00331* (2015).
- [13] S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, K.S. Candan, Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents, in: Proceedings of the 32nd International Conference on Very Large Data Bases, 2006, pp. 283–294.
- [14] J. Li, J. Wang, Fast matching of twig patterns, in: Proceedings of the 19th International Conference on Database and Expert Systems Applications, 2008, pp. 523–536.
- [15] T. Yu, T.W. Ling, J. Lu, TwigStackList: a holistic twig join algorithm for twig query with not-predicates on XML data, in: Proceedings of the 11th International Conference on Database Systems for Advanced Applications, 2006, pp. 249–263.
- [16] J. Lu, T.W. Ling, C.Y. Chan, T. Chen, From region encoding to extended Dewey: on efficient processing of XML twig pattern matching, in: Proceedings of the 31st International Conference on Very Large Data Bases, 2005, pp. 193–204.
- [17] T. Chen, T.W. Ling, C.Y. Chan, Prefix path streaming: a new clustering method for optimal holistic XML twig pattern matching, in: Proceedings of the 15th International Conference on Database and Expert Systems Applications, 2004, pp. 801–810.
- [18] A. Wojnar, I. Mlýnková, J. Dokulil, Structural and semantic aspects of similarity of document type definitions and XML schemas, *Inf. Sci.* 180 (10) (2010) 1817–1836.
- [19] S. Lee, B. Ryu, K. Wu, Examining the impact of data-access cost on XML twig pattern matching, *Inf. Sci.* 203 (2012) 24–43.
- [20] W. Zuo, Y. Chen, F. He, K. Chen, Load balancing parallelizing XML query processing based on shared cache chip multi-processor (CMP), *Sci. Res. Essays* 6 (18) (2011) 3914–3926.
- [21] L. Shnaiderman, O. Shmueli, Multi-core processing of XML twig patterns, *IEEE Trans. Knowledge Data Eng.* 27 (4) (2015) 1057–1070.
- [22] S. Staworko, P. Wiczorek, Characterizing XML twig queries with examples, in: Proceedings of the 18th International Conference on Database Theory, 2015, pp. 144–160.
- [23] J. Liu, Z.M. Ma, Q. Qv, Dynamically querying possibilistic XML data, *Inf. Sci.* 261 (2014) 70–88.
- [24] L. Bai, Y. Li, J. Liu, Fast leaf-to-root holistic twig query on XML spatiotemporal data, *J. Comput.* 12 (6) (2017) 534–542.
- [25] L. Bai, Y. Li, J. Liu, Fspwfast: holistic twig query on fuzzy spatiotemporal XML data, *Appl. Intell.* 47 (4) (2017) 1224–1239.
- [26] Z.M. Ma, J. Liu, L. Yan, Matching twigs in fuzzy XML, *Inf. Sci.* 181 (1) (2011) 184–200.
- [27] P. Buneman, G. Cong, W. Fan, A. Kementsietsidis, Using partial evaluation in distributed query evaluation, in: Proceedings of the 32nd International Conference on Very Large Data Bases, 2006, pp. 211–222.
- [28] G. Cong, W. Fan, A. Kementsietsidis, Distributed query evaluation with performance guarantees, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2007, pp. 509–520.
- [29] I. Machdi, T. Amagasa, H. Kitagawa, Executing parallel TwigStack algorithm on a multi-core system, in: Proceedings of the 11th International Conference on Information Integration and Web-based Applications and Services, 2009, pp. 176–184.
- [30] X. Bi, X. Zhao, G. Wang, Efficient processing of distributed twig queries based on node distribution, *J. Comput. Sci. Technol.* 32 (1) (2017) 78–92.
- [31] H. Lee, W. Lee, Selectivity-sensitive shared evaluation of multiple continuous XPath queries over XML streams, *Inf. Sci.* 179 (12) (2009) 1984–2001.
- [32] W. Hao, K. Matsuzaki, A partial-tree-based approach for XPath query on large XML trees, *J. Inf. Process.* 24 (2) (2016) 425–438.
- [33] M. Damigos, M. Gergatsoulis, E. Kalogeros, Distributed evaluation of XPath queries over large integrated XML data, in: Proceedings of the 18th Panhellenic Conference on Informatics, 2014, pp. 61:1–61:6.
- [34] J. Son, H. Ryu, S. Yi, Y.D. Chung, SSFile: a novel column-store for efficient data analysis in Hadoop-based distributed systems, *Inf. Sci.* 316 (2015) 68–86.

- [35] R. Hricov, A. Senk, P. Kroha, M. Valenta, Evaluation of XPath queries over XML documents using SparkSQL framework, in: Proceedings of 13th International Conference on Beyond Databases, Architectures and Structures, 2017, pp. 28–41.
- [36] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, D. Srivastava, Structural joins: a primitive for efficient XML query pattern matching, in: Proceedings of the 18th International Conference on Data Engineering, 2002, pp. 141–152.
- [37] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, in: Proceedings of the 27th International Conference on Very Large Data Bases, 2001, pp. 361–370.
- [38] I. Tatarinov, S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita, C. Zhang, Storing and querying ordered XML using a relational database system, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002, pp. 204–215.
- [39] B. Ning, C. Liu, XML filtering with XPath expressions containing parent and ancestor axes, *Inf. Sci.* 210 (2012) 41–54.
- [40] W. Hsu, I. Liao, CIS-X: A compacted indexing scheme for efficient query evaluation of XML documents, *Inf. Sci.* 241 (2013) 195–211.
- [41] D. Gopinathan, K. Asawa, New path based index structure for processing CAS queries over XML database, *Comput. Inf. Technol.* 25 (3) (2017) 211–225.
- [42] G.Z. Qadah, Indexing techniques for processing generalized XML documents, *Comput. Stand. Interfaces* 49 (2017) 34–43.