

An Efficient Parallel Approach of Parsing and Indexing for Large-scale XML Datasets

Kunfang Song, Hongwei Lu*

School of Computer Science & Technology
Huazhong University of Science and Technology
Wuhan, China
skf@wtu.edu.cn, luhw@hust.edu.cn

Xiao Qin

Computer Science and Software Engineering
Auburn University
Auburn, American
xqin@auburn.edu

Abstract—MapReduce is a widely adopted computing framework for data-intensive applications running on clusters. We propose an approach to exploit data parallelisms in XML processing using MapReduce in Hadoop. Our solution seamlessly integrates data storage, labelling, indexing, and parallel queries to process a massive amount of XML data. Specifically, we introduce an SDN labelling algorithm and a distributed hierarchical index using DHTs; we develop an efficient data retrieval approach called B-SLCA. More importantly, we design an advanced two-phase MapReduce solution that is able to efficiently address the issues of labelling, indexing, and query processing on big XML data. We implemented our solution on a real-world Hadoop cluster processing the real-world datasets. Our experimental results show that SDN outperforms NCIM by up to a factor of 1.36 with an average of 1.17; our B-SLCA outperforms BwdSLCA by up to a factor of 1.96 with an average of 1.2.

Keywords—Big XML; Parallel Programming; Distributed Programming; MapReduce; SDN; B-SLCA;

I. INTRODUCTION

XML processing has been extensively studied in the literature. During the past few years, a handful of efficient approaches have been proposed for the XML processing evaluation, which includes three key operation parts, labelling [4], [10], [13], indexing [2], [9], [1], [10], and keyword searching [8], [7]. Much attention has been paid to keyword searching on XML data. Efficient query processing of large XML data plays an important role in keyword searching, because various applications depend on fast query performance to simultaneously support an enormous number of users. The most widely adopted semantics are arguably the smallest lowest common ancestor or SLCA [21], [20], [14], [3] and exclusive lowest common ancestor or ELCA. In addition, a few efficient XML data processing schemes found in the literature include pipeline [22], DHT [17], multi-code [6], [15] and the like. Growing evidence shows that indexing techniques boost XML processing performance, because data are accessed via indices rather than scanning an entire data.

The XML operator typically includes labelling, indexing, and keywords searching, among which labelling and

indexing are two important components. Since semantics are defined using the notion of lowest common ancestor (LCA), at the heart of existing query algorithms is the Dewey labelling [13]. The Dewey label of a node u is a concatenation of all its ancestor nodes' local label on the path from the document root to u . Much attention has been paid to keywords searching on XML files. It is demanding to design efficient query processing methods for keyword searching on XML data, because XML applications require fast query performance to meet the needs of a large number of users.

This study is focused on XML files that adopt the standard XML format, where each file is characterized as an ordered, rooted, and labelled tree [16]. Each edge represents an element-element relationship or an element-value relationship. Each element is identified by a pair of start-tag and end-tag; elements may have attributes with their values. If keyword k appears at least once in one of a node name, an attribute name, and text value of root node v , we say v directly contains k . Figure 1 illustrates a sample of standard XML document.

Compared with the parallel interval-based algorithm [4][10], our solution has the following salient features. First, we consider offsets of nodes in labels and use the offsets to implement keywords query processing. Second, we propose hash-based search algorithms to accelerate the B-SLCA computation. After quantitatively comparing our solution and the existing methods, we discuss the extension of our approach to other LCA-based semantics.

More concretely, the contributions of this paper can be summarized as follows:

- We develop the SDN labelling technique for each element in *Hadoop distributed file system* or HDFS.
- We construct a flexible indexing model based on DHTs, thereby improving query performance of XML datasets stored in HDFS. The DHT-based indexing model avoids repeatedly scanning an entire dataset without pruning distance computations. We implement DHT index using MapReduce when processing the large XML datasets.

- We develop the B-SLCA keyword search approach using SDN label in DHTs. B-SLCA is a bottom-up retrieval way to quickly discover an SLCA node. We use the offsets of nodes to locate a result block in an entire XML document.

The rest of this paper is organized as follows. Section 2 describes background knowledge and preliminaries of this study. In section 3, we introduce an overview of the labelling and indexing model in the MapReduce framework. XML data partition processing, SDN labelling, and B-SLCA search algorithms are discussed in Section 4. Section 5 discuss the experimental results and compares with the existing algorithms, and performance analysis. We presents the conclusions in Section 6.

II. PRELIMINARIES

In this section, the MapReduce programming framework can be briefly reviewed. Then, we introduce an SDN labelling and a DHT-based indexing model to transform the structured XML file to sequence index data.

A. The MapReduce Framework

MapReduce [5], [12] has emerged as a popular and easy-to-use programming framework to support parallel data-intensive applications processing explosive amounts of data. In MapReduce, input data are represented as key-value pairs. A group of functional programming primitives including Map and Reduce are introduced to process the data.

Programming model. An application following the MapReduce programming model is mainly composed of a map function and a reduce function along with data types. In the first indexing phase of our system, map and reduce implement the following signatures:

$$\begin{aligned} \text{map} &:: (K_1, V_1) \rightarrow [(K_2, V_2)] \\ \text{reduce} &:: (K_2, [V_2]) \rightarrow [(K_3, V_3)] \end{aligned}$$

The MapReduce framework divides the input data into kv-pairs, and splits a potentially large list into smaller lists. The details of generating kv-pairs as well as input splits can be specified by users via a custom split function. After kv-pairs are created and partitioned into input splits, the framework will apply one separate map process for each input split. Map processes are typically spawned on different machines to take advantage of parallel resources. Similarly, multiple reduce processes are configured to concurrently process different distinct keys produced by the map processes.

Hadoop. Hadoop is an open source software that implements the MapReduce framework. Data in the Hadoop runtime system are managed by HDFS, which consists of (1) multiple DataNodes storing data and (2) a master node monitoring DataNodes and maintaining metadata. In HDFS, imported data are split into equal-size chunks, which are allocated by NameNode to multiple DataNodes.

The Hadoop runtime system consist of two processes, namely, JobTracker and TaskTracker. After splitting a submitted job into map and reduce tasks, the JobTracker schedules the tasks among all available TaskTrackers, which process the assigned map/reduce tasks. For each map task, the TaskTracker takes a data chunk specified by the JobTracker and then applies the map function. When all the map() functions complete, the runtime system groups all intermediate results and launches a number of reduce tasks to run the reduce function producing final results.

B. Labelling and Indexing

1) *SDN Labelling*: In order to process XML queries, one has to identify relationships among elements (e.g., ancestor-descendant and parent-child relationships). In this paper, we introduce a tree labelling algorithm to discover a relationship between two XML elements by simply comparing two labels associated with the elements. The labelling algorithm aims to improve the query processing performance by serving queries without traversing an entire XML tree.

Definition (SDN Labelling) Interval-based encoding represents the position of an XML tree node ν in document order as a 5-tuple $(start, end, depth, offset, pstart)$, where a pair of $start$ and end indicates an interval of ν , $depth$ denotes the level of node ν , $offset$ represents the position of node ν in XML tree D , and $pstart$ denotes the parent's start position of ν .

For any two XML tree nodes u and ν , u is a parent of node ν if fu is an ancestor of ν and $\nu.level = u.level + 1$. Also, node u precedes node ν in a document order if $fu.start < \nu.start$. First, we employ a semantic partitioning method to split an entire big XML document. Next, we label each node based on SDN using the map function. Last, we gather all the sorted labelling nodes in the reduce function to produce an index file.

2) *DHT-based Inverted Indexing*: DHT, a data retrieval approach, is implemented in a decentralized distributed system that provides a lookup service similar to a hash table. $(key, value)$ pairs are stored in a DHT, where any participating node efficiently retrieves a value associated with a given key. In this study, key denotes a node's tag name and $value$ represents a pre-labelled value. Mapping from keys to values is locally and independently managed by each node; in doing so, a change in the set of participants causes little disruption. All keys are formally expressed as $Q = \{k_1, k_2, \dots, k_m\}$; corresponding values are denoted as $L = \{v_1, v_2, \dots, v_n\}$.

C. Query Semantics

Given a query $Q = K_1, K_2, \dots, K_m$ and an XML document D , we denote R_i as an inverted list of K_i containing nodes elements. All the labels of R_i are sorted in the document order. Let $LCA(\nu_1, \nu_2, \dots, \nu_m)$ be the

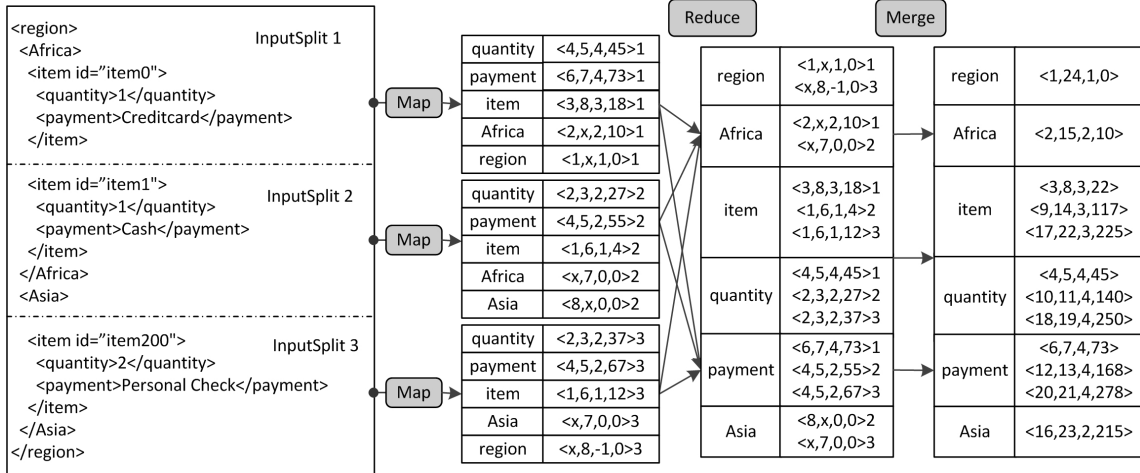


Figure 1: An example of the SDN labelling process

lowest common ancestor (LCA) of nodes $\nu_1, \nu_2, \dots, \nu_m$, the LCAs of Q on D are expressed as $LCA(Q) = \{V | V = LCA(\nu_1, \nu_2, \dots, \nu_m), \nu_i \in R(1 \leq i \leq m)\}$. Based on the LCA semantics, the most widely adopted variants are SLCA [21]. SLCA defines a subset of $LCA(Q)$, of which no one is the ancestor of any other LCA formally defined as follows.

Definition (SLCA) Given a keyword query $Q = \{k_1, k_2, \dots, k_m\}$ and an XML document D , the set of SLCA of Q on D are $SLCA(Q) = \{\nu | \nu \in LCA(Q) \cap \nexists \nu' \in LCA(Q)\}$.

III. SDN LABELLING AND INVERTED INDEXING

Figure 1 outlines the basic idea of our parallel XML processing techniques using MapReduce in two steps. First, an input XML document is loaded from HDFS. During the loading procedure, a large XML document is split into multiple 64-MByte blocks. Second, each map task is responsible for labelling by employing the SDN signature method for XML elements in an InputSplit. A challenge lies in the fact that a start-tag and its corresponding end-tag may not be placed together in the same chunk. To address this problem, we exploit an additional split information table called *SIT* to build complete indices for discrete tags in blocks, which record mismatching tags and their levels in the blocks.

Definition (Split Information Table, SIT) The table represents a 3-tuple $(splidId, count, level)$, which is stored in the form of a hash table with each chunk's informations in HDFS. In the map phase, the table will record the tags count, mismatching tags count, and its level in each chunk, respectively; in the reduce phase, the table is applied to manage the counts and depth of previous InputSplit continuously according to the merge processing method.

Recall that all steps of our hashing-based SDN labelling, inverted indexing, and query processing can be easily distributed. The first step of our system is to perform the hash tables construction to build indices in each blocks; the second step is to implement keywords query, which includes two MapReduce phases – gathering all the elements of keywords and querying data blocks based on results yielded from the first phase.

A. Data partitioning

It is worth noting that splitting an XML document is challenging because of its open semi-structured textual format. Ideally, partitioning an XML document should guarantee data integrity, which means that XML segments partitioned by data placement strategy should not change the semantic meaning of original the XML document. Unfortunately, the default *InputFormat* class in Hadoop reads and splits texts line-by-line; the *TextInputFormat* class is inadequate for processing XML documents due to the lack of XML semantics in the class, which splits an input file simply on the basis of InputSplit size. For example, suppose that there is an XML document containing text $\langle author \rangle Jack \langle /author \rangle$, which is written in two lines (e.g., l_i and $l_i + 1$). If reaching the threshold on individual InputSplit size (the default is 64M) is just at the end of line l_i , node semantic cannot be guaranteed.

B. SDN labelling

The *XMLInputFormat* class splits input XML document D into independent chunks of approximately equal sizes (e.g., the default split size) when uploading XML files from HDFS. Our parallel indexing scheme includes two phases: partial encoding in the map task and creating indices in the reduce task. A mapper iterates over each line of a chunk and a encodes tag name for each element. When identifying a correct pair of tag ν , our parallel encoding scheme

Algorithm 1 SDN Labelling

Input:

```
1: Stack Stag = [];  
2: Hash Table HT = [];  
3: Long tagCount = 0L; /*tagC*/  
4: Long tagDepth = 0L; /*tagD*/  
5: Long splitId = 0L;  
6: Long tagOffset = 0L; /*tagO*/  
7: Long pareOffset = 0L; /*pareO*/  
8: String tagName = null;  
9: String tagContent = null;  
10: L = [tagS, tagE, tagD, tagO, pareO];  
11: SplitTable = [splitId, Count, Level];
```

Output:

```
12: splitId = context.getInputSplit().getStart();  
13: function MAP(LongWritable key, Text value)  
14:   tagCount = tagCount + 1;  
15:   tagName = value;  
16:   tagOffset = key.get();  
17:   if tag is a start-tag then  
18:     tagDepth = tagDepth + 1;  
19:     new L = (tagC, x, tagD, tagO, -100L);  
20:     if !Stag.empty() then  
21:       L.pareOffset(Stag.peek().TagOffset);  
22:     end if  
23:     Stag.push(L);  
24:   else /* If tag is an end-tag */  
25:     tagDepth = tagDepth - 1;  
26:     new L = (x, tagC, tagD, tagO, -100L);  
27:     output (tagName, L) into HT;  
28:     if !Stag.empty() then  
29:       L = stack.pop();  
30:       L.tagEnd = tagCount;  
31:       output (tagName, L) into HT;  
32:     end if  
33:   end if  
34: end function  
35: output SplitTable ST = [splitId, Count, Level];
```

computes ν 's start position (i.e., $\nu.s$), end position (i.e., $\nu.e$), level (i.e., $\nu.l$), and start-tag's offset (i.e., $\nu.o$). If the tag is a leaf node, then $\nu.e$ is derived from $\nu.s$ as $\nu.e = \nu.s + 1$. We annotate each node with a 5-tuple of integer numbers $\langle tagStart, tagEnd, tagDepth, tagOffset, pStart \rangle$ as the pre-encode of value in the map phase. Then, each value is emitted along with a table identifier (i.e., tag name). We call the 5-tuple sequence depth number labelling or SDN for short. The Reduce function merges all the emitted identifiers sharing the same tag name. Resulting buckets are provided as an index table that facilitates the process of the second MapReduce job.

The inputs of our labelling algorithm are a list of

key-value pairs, each of which is in the form of $\langle InputSplitId, atag \rangle$. During the initialization phase, two variables *currentCount* and *currentLevel* are set to zero; two object stacks and offset recorder are initialized to null. The *currentCount* value of each tag increases by one. If a tag is a start-tag, the algorithm builds a new label with a start value set to *currentCount*; then, the encoded data is pushed into stack *S*. If a tag is an end-tag, then the map function (1) pops an encode data from *S*, (2) sets an end value in the encoded data to *currentCount*, and then (3) emits the encoded data as an output. The *currentLevel* value is used to quantify the level of an element. While processing a start-tag, the algorithm increases *currentLevel* by one; the encoded data along with the start value are pushed into *S*. On the contrary, the algorithm decreases *currentLevel* by one for each end-tag. The algorithm is an extension of the conventional interval-based encoding scheme; our encoding scheme is different from the traditional one in that ours handles open-started and open-ended tags. A label is considered as an open-started one if its start value is unknown (e.g., (x, 7, 0, 0, x)). An open-started label is encoded when our algorithm encounters an end-tag that is not paired by a corresponding start-tag in the same InputSplit. This case occurs if a start-tag and its end-tag are located in two separate InputSplits. Similarly, a label is marked as an open-ended label if the end value of the label is unknown (e.g., (1, x, 1, 0, -1)). Algorithm 1 describes the partial encoding process implemented in the map function.

An open-ended label is encoded when the algorithm encounters a start-tag that does not have a corresponding end-tag in the same InputSplit. If the algorithm sees an end-tag that does not have its start-tag in the same InputSplit, a label is popped up from *S* even though *S* is empty. In that case, the map function emits an open-started label. On the other hand, if there are open-start-tags (i.e., no corresponding end-tags in the same InputSplit), some open-ended labels will still be left in *S* after the the map function is completed. This problem is solved by forcing the map function to emit all open-ended labels from *S* at the end of the map phase. Note that the *currentLevel* value may be negative as an InputSplit may have multiple end-tags, which are not paired by corresponding start-tags in the same InputSplit. In addition, it is noteworthy that each key in mapper outputs is a composite key that is composed of the tag name, InputSplit Id, and the start value. Such a composite key is beneficial for both grouping and sorting mapper outputs. With the composite key scheme in place, mapped outputs are grouped by distinct tag name. Then, records in each group are sorted by InputSplit Id and the start value. Note that this grouping and sorting are naturally implemented in a grouping strategy that extends the MapReduce merge sort [11]. Finally, each map task writes the final state of the *currentCount* and *currentLevel* values into HDFS.

Each map task stores state information *ST* by recorder,

which consists of the tags count's final state, tags mismatching count, and offset with respect to its InputSplit. *offset* represents the offset of start-tag in the InputSplit of a block, which is a key part of the query in the second MapReduce job.

C. DHT-based inverted indexing

The purpose of this phase is three fold: (1) to merge map results, (2) to upgrade encoded data with each InputSplit except the first one, and (3) to build a complete index list of a big XML document. Algorithm 1 describes the pseudocode of our proposed labelling process, which facilitates partial encoding. First, the labelling algorithm builds a table (i.e., *ST*) providing information that is required to handle partial coding at the reduce stage. An *ST* consists of three columns: InputSplitId, a cumulative count value, and a cumulative level value. Each row ST_i in table *ST* is computed as

$$ST_{i,p} = \begin{cases} 0 & \text{if } i = 1 \\ \sum_{k=1}^{i-1} L_{k,p} & \text{if } i \neq 1 \end{cases} \quad (1)$$

$(p = Count, Depth, Offset)$

Next, the labelling algorithm initializes a stack (i.e., *S*), which aims at pairing open-ended labels to their corresponding open-started encoding. Note that at the reduce stage, input data are grouped by tag names and also sorted in an ascending order of $(inputSplitId, tagStart)$, and the order is determined by the composite key scheme. As a result, each reduce function manages sorted encoding lists, each of which represents elements sharing the same tag name. The reduce() function deals with the following three cases.

- When the reduce function sees label *L* that is neither open-ended nor open started, the reducer simply adds $ST_i.tagCount$ to both $L.tagStart$ and $L.tagEnd$. Also, the reducer adds $ST_i.tagDepth$ to $L.tagDepth$. It is noteworthy that table *ST* makes it possible to keep track of the cumulative tag-count, mismatching tag-count, and offset for each InputSplit. Therefore, by simply adding the values in ST_i to labels for the elements in the *i*th InputSplit, the labels are completed just as if all XML elements were numbered in serial from the beginning.
- When the reducer encounters an open-ended label *L*, the reducer adds $ST_i.tagStart$ and $ST_i.tagDepth$ to $L.tagStart$ and $L.tagDepth$, respectively. Then, *L* is pushed into *S*.
- When an open-started label *L* appears, the reducer adds $ST_i.tagEnd$ and $ST_i.tagDepth$ to $L.tagEnd$ and $L.tagDepth$, respectively. The reducer pops an open-ended label *L* from *S* then merges *L* and an open-started label *L* into a single label *L* by a label-merge operator $\oplus$ defined as below.

Definition (Offset Recorder) Given an open-started code *L* and an open-ended label *L'*, the offset recorder operator denoted as $\oplus$ is defined by

$$L \langle x, tagEnd, tagDepth_1, tagOffset_1, pareOffset_1 \rangle \oplus L' \langle tagStart, x, tagDepth_2, offset_2, pareOffset_2 \rangle \rightarrow L \langle tagStart, tagEnd, tagDepth_2, pareOffset_1 \rangle$$

For example, the first reducer reads the open-ended label $L' \langle 2, x, 2, 10, 1 \rangle$ for the 'Africa' element in the first InputSplit. Then, the reducer adds values in L_1 to *L*, which is pushed into stack *S*. No change occurs in *L* because values in L_1 are all zero. Next, the reducer reads open-started label $L' \langle x, 7, 0, 0, x \rangle$ from the second InputSplit, followed by adding values in T_2 to *L* (i.e., $L' \langle x, 7 + 8, 0 + 2, 10, 0 \rangle$). Finally, the reducer pops *L* from *S* so that $L' \langle 2, x, 2, 10, 0 \rangle$ and $L' \langle x, 15, 2, 10, 0 \rangle$ are merged into a single label $L' \langle 2, 15, 2, 10, 0 \rangle$.

SID	count	level	offset
1	8	2	113
2	8	0	108
3	8	-2	119

The offset recorder of Map function

SID	count	level	offset
1	0	0	0
2	8	2	221
3	16	2	340

The offset recorder of Reduce function

Figure 3: The Offset Recorder of first MapReduce

Each reducer manages sorted label lists grouped by tag names; each reducer always keeps open-ended tags in *S* in an ascending order of start values for a single tag name. Therefore, whenever an open-ended tag is popped from *S*, the current open-started tag pairs with its corresponding open-ended tag. Fig. 3 shows the *OffsetRecorder* of the map task and reduce task in this example.

IV. B-SLCA: A MAPREDUCE-BASED KEYWORD SEARCH QUERY

In this work, we compute the SLCA value of all keywords by taking the prefix of values in an index table produced by the first MapReduce job. Assume that there are *n* keywords $(k_1, \dots, k_n)$ in query statement *Q*, there must be a vector $K = (k_1, \dots, k_n)$ that matches the keywords in query, in which k_i is an arbitrary match in element *n* with respect to keyword K_i . A resulting subset denoted as *G* is composed of all data items that key is keywords k_i and the label value is the selected object.

A. Data Grouping

If query *Q* has *k* keywords, the group must contain *k* items with each keyword. We set up a vessel with the size of *k* like queue *Qu*, and then load the items into the queue in a proper order. Our group mechanism decides whether there exists the same tag name *u* in *Qu* before loading. There are three types of decisions when an item is placed in a queue.

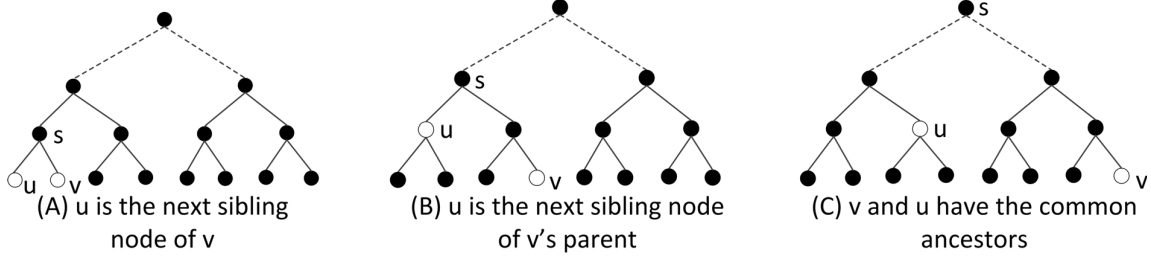


Figure 2: Positional relationships between nodes u and v in group g

- If the same tag u are not found in Qu , we will put u in Qu . All items in queue may form a group or there exists the same tag name in the queue when the queue is full.
- If there exists the same tag u in Qu and the queue is not full, it means that all items before u (u is included) are mismatched nodes and should be filtered. This process is repeatedly executed until queue Qu is full.
- If there is the same tag u in Qu and the queue is full, The items in queue may form a group if it includes every keyword; then, the queue becomes empty.

When the queue is full and the same tag name is not found, the B-SLCA approach determines the items of a group according to the regulations (see the next section). Our approach is efficient for a standard XML tree like an extensible bivariate data table.

B. Obtaining SLCA Nodes

Let us consider a grouped dataset $G = G_1, \dots, G_m$, each of which is a data group (i.e., $G_j = L1, L2, \dots, L_n$) that includes keywords $k_1, \dots, k_n$. We denote $MIN_j depth$ as the minimum depth value of all the items in G_j ; thus, we can get the value by $MIN_j depth = MIN_{1 \leq i \leq n} L_i.depth$. The goal is to find the smallest lowest common $pareOffset$ of all the items in each group G_i . If so, the current level of grouped G is not the same level generally, and we write a recursive function to obtain the common ancestor of G by $G_j.pareOffset$ onto auxiliary indices as well as getting a data block in the XML tree.

1) *Backtracking Parameter λ* : Recall that we take the bottom-up search strategy to traverse keywords from leaves back to the root of an XML tree; the search strategy is implemented in the form of a recursive function, where parameter λ specifies the number of recurrences.

To characterize k_i 's relationship with respect to its group, we conduct descriptive and correlational analysis; we then quantify k_i 's distance according to parameter λ . The value of the λ depends on the XML structure and depth. The λ value is very small (i.e., lower level) when the XML file has a stable structure. For complicated and multilevel files, choosing an appropriate λ value is critical.

2) *The B-SLCA Search Approach*: Given two nodes u and v in a group, the relationship between the two nodes can be summarized as follows.

- Case 1: Fig. 2 (A), u is the next sibling node of v . The two nodes have the same smallest lowest common parent $offset$ (i.e., $u.pareOffset = v.pareOffset$).
- Case 2: Fig. 2 (B), u is the next sibling node of v 's parent.
- Case 3: Fig. 2 (C), u and v have a common ancestor; we have to find the SLCA node.

When parameter λ is small (e.g., $\lambda = 1$ or 2), it is trivial to find an SLCA node for cases 1 and 2. In case 3, u and v may be either two distant relatives or have no relationship at all. We control the cycle count using a recursive function by parameter λ to improve the performance. For general cases, it is time consuming, if not impossible, to find an SLCA node (see Fig. 2 (C)).

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

In our experiments, we set the sizes of four big XML files to 10GB, 20GB, 30GB, and 40GB, respectively; times spent in making labels and building indices for the four big XML files are 934s, 2173s, 3085s, and 3943s, respectively. Our preliminary findings show that the average depth is 5.

We create four groups of queries to evaluate our algorithm processing XML data of various sizes. The Xmark dataset is tested in our experiments, because the Xmark dataset allows us to gain insights into the characteristics of XML repositories. After parsing the data, the XML file is split into a set of 64MB HDFS blocks. Each data node starts labelling every keyword using the SDN algorithm. Then, the hash table stores the labelled keywords in the form of an inverted list. Fig. 5 shows that with the increase of the result selectivity, the average overall response time in different queries.

B. Indexing Time Comparison

As an important step of large-scale XML datasets indexing, which include parsing, SDN labelling and indexing, we compared our SDN approach with existing efficient method, NCIM(Node Clustering Indexing Method [19]) and

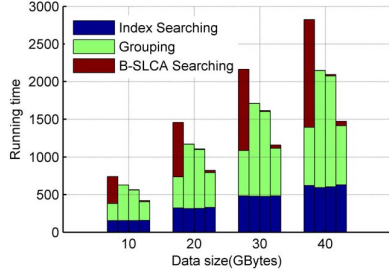


Figure 4: Total running time

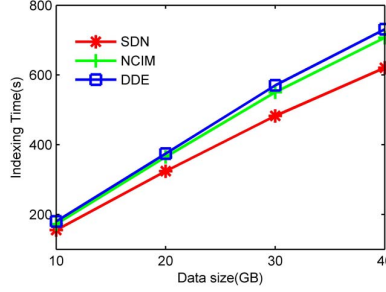


Figure 5: Indexing time

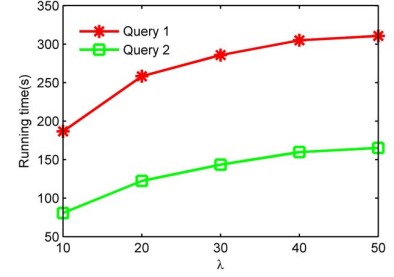


Figure 6: Impact of λ on running time of B-SLCA

DDE(for Dynamic DEwey [13]). The label format of DDE is a sequence of components separated by '.', which inherits the Dewey encoding rules. The NCIM method labels each element node of an XML data tree with 3-tuple(level, m^* , n^*) for non-leaf and a 2-tuple(level, m^*) for leaf node. It is difficult to discriminate the relationship of any two nodes in NCIM method. Our excellent performance can be explained according to Fig. 4, which shows that SDN outperforms NCIM by up to a factor of 1.36 with an average of 1.17.

C. Impact of Backtracking Times of λ

Parameter λ is applied to reduce the cycle times in the B-SLCA processing, especially in typical XML trees with extremely small depth. In this section, we study the impact of λ on the performance of our SLCA-node-searching techniques. While identifying the SLCA node of a group's items, we control the backtracking times and to optimize system performance and outcome. In our approaches, we adopt the nearest relationship principle to find the SLCA nodes. Fig. 6 plots the results when λ is varied from 1 to 5 on query 1 and query 2.

Fig. 6 shows that when λ increases, the running time of B-SLCA goes up accordingly. After λ is larger than 4, the running time becomes insensitive to λ . Thus, the maximum value of λ is the maximum depth of the XML files. This performance result indicates that the λ plays an important role in searching SLCA nodes when the depth value is large. After testing the result error rate with various λ values, we observe that the maximum error rate is 6 on group 1 in the 4 GB XML file. We conclude from Fig. 6 that when λ increases, the B-SLCA processing cost remains unchanged. The result suggests that it is important to optimize the λ value after we characterize the patterns of the XML files.

D. Multi-job Parallel Query

Now, we focus on multi-query parallel processing to evaluate the effectiveness of our optimization algorithms running on Hadoop clusters. We compared our solution with the existing BwdSLCA [10] and GGT [18] algorithms. We conduct the experiments by repeatedly running the query process in group 4, where average query performance is reported.

Impact of the number of queries. Fig.7 shows the impact of the size of a query batch on system performance. The experimental results confirm that our algorithm significantly outperforms BwdSLCA and GGT. For example, when the number of queries is set to 25, B-SLCA outperforms GGT by up to a factor of 1.65 with an average of 1.14; our B-SLCA outperforms BwdSLCA by up to a factor of 1.96 with an average of 1.2. Furthermore, as the number of queries increases, the winning margin of our algorithms over BwdSLCA increases. This performance trend is expected, because the sharing opportunities among queries increase with the growing query batch size.

Impact of cluster size. Fig. 8 plots the scalability of our B-SLCA by evaluating the impacts of the number of data nodes in the Hadoop cluster. The experimental results are consistent with those shown in the previous experiments; our B-SLCA is superior to GGT and BwdSLCA. For example, when the number of data nodes is eight, GGT outperforms NA by up to a factor of 1.56 with an average of 1.22. Furthermore, as the Hadoop cluster scales up, the running time for all the algorithms decreases. In particular, the running time of BwdSLCA decreases much faster than those of the other algorithms; therefore, the winning margin of B-SLCA is reduced as cluster size increases. Thus, a large Hadoop cluster is more beneficial to the non-optimized NA than the optimized non-NA counterparts.

Fig. 9 reveals that the gap of running time among the three datasets tends to be narrowed when the number of data nodes increases, and the real-world dataset is deliberate tactic to evaluate and contrast our algorithms with the other datasets.

VI. CONCLUSION

A performance bottleneck for parallel XML processing lies in the keywords query operation, which is commonly used to offer historical data analysis. We designed a systematic XML data processing called B-SLCA. Rather than applying Dewey labels, our labelling scheme treats XML data nodes' position, depth, and offset as labels. The inverted indices are built with SDN labels based on DHTs. We speed up the query processing by the optimization methods based on the query semantics and structural relationship among

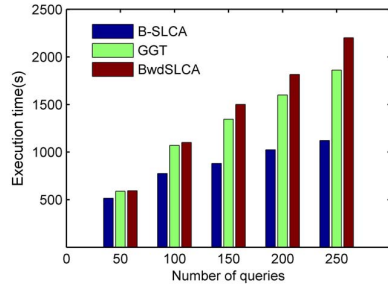


Figure 7: Impact of the number of queries

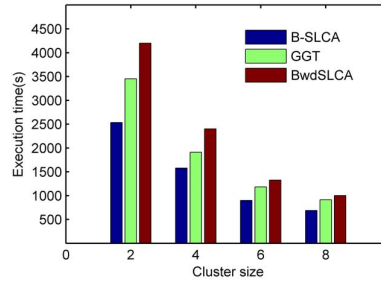


Figure 8: Impact of cluster size

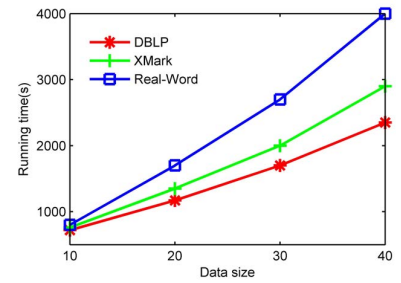


Figure 9: Impact of data size

nodes. Furthermore, we proposed a hash search scheme to reduce boost query performance.

REFERENCES

- [1] A. Aranda-Andjar, J. Camacho-Rodriguez, D. Colazzo, F. Goasdou, and Z. Kaoudi. Amada: web data repositories in the amazon cloud. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2749–2751, 2012.
- [2] G. Chen, H. T. Vo, and B. C. Ooi. A framework for supporting dbms-like indexes in the cloud. In *VLDB*, volume 4, pages 702–713, 2011.
- [3] L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, 2010.
- [4] H. Choi, K.-H. Lee, and Y.-J. Lee. Parallel labeling of massive xml data with mapreduce. In *The Journal of Supercomputing*, 2013.
- [5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
- [6] Z. Fadika, M. R. Head, and M. Govindaraju. Parallel and distributed approach for processing large-scale xml datasets. In *10th IEEE/ACM International Conference on Grid Computing*, pages 105–112, 2009.
- [7] J. Feng and G. Li. Efficient fuzzy type-ahead search in xml data. In *IEEE Transactions on Knowledge and Data Engineering*, volume 24, pages 882–895, 2012.
- [8] J. F. G. Li, C. Li and L. Zhou. Sail: Structure-aware indexing for effective and progressive top-k keyword search over xml documents. In *Information Sciences*, volume 179, pages 3745–3762, 2009.
- [9] W.-C. Hsu, I.-E. Liao, and H.-C. Shih. A cloud computing implementation of xml indexing method using hadoop. In *Intelligent Information and Database Systems*, volume 7198, pages 256–265, 2012.
- [10] W. W. J. Z. Junfeng Zhou, Zhifeng Bao and X. Meng. Efficient query processing for xml keyword queries based on the idlist index. In *The VLDB Journal*, pages 1–26, 2013.
- [11] L. Y. C. Y. Lee K, Choi H and M. B. Parallel data processing with mapreduce: A survey. In *ACM SIGMOD Record*, volume 40, page 1120, 2012.
- [12] U. G. Leonidas Fegaras, Chengkai Li and J. J. Philip. Xml query optimization in map-reduce. In *WebDB*, 2011.
- [13] H. W. Liang Xu, Tok Wang Ling and Z. Bao. Dde: from dewey to a fully dynamic xml labeling scheme. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 719–730, 2009.
- [14] Y. Ling and G. Xu. A distributed keyword search algorithm in xml databases using mapreduce. In *Computer, Informatics, Cybernetics and Applications*, volume 107, pages 1307–1316, 2012.
- [15] W. Lu and D. Gannon. Paraxml: A parallel xml processing model on the multicore cpus. In *Technical Report*, 2008.
- [16] L. Q and M. B. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th VLDB conference*, 2001.
- [17] N. P. N. P. S. Abiteboul, I. Manolescu and C. Sun. Xml processing in dht networks. In *ICDE 2008*, 2008.
- [18] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. In *VLDB*, volume 7, pages 145–156, 2014.
- [19] I.-E. L. Wen-Chiao Hsu and H.-C. Shih. A cloud computing implementation of xml indexing method using hadoop. In *Intelligent Information and Database Systems*, volume 7198, pages 256–265, 2012.
- [20] C. Zhang, Q. Ma, X. Wang, and A. Zhou. Distributed slca-based xml keyword search by map-reduce. In *Database Systems for Advanced Applications*, volume 6193, pages 386–397, 2010.
- [21] M. Zhou, H. Hu, and M. Zhou. Search xml data by slca on a mapreduce cluster. In *IUCS*, pages 84–89, 2010.
- [22] D. Zinn, S. Bowers, S. Kohler, and B. Ludascher. Parallelizing xml data-streaming workflows via mapreduce. In *Journal of Computer and System Sciences*, volume 76, page 447463, 2010.