J Supercomput
DOI 10.1007/s11227-013-1008-6

Parallel labeling of massive XML data with MapReduce

Hyebong Choi · Kyong-Ha Lee · Yoon-Joon Lee

Abstract The volume of XML data has become enormous and still grows very quickly as many data have been typed in XML by virtue of its simplicity and extensibility. While a tree labeling algorithm has a crucial role in XML query processing, conventional algorithms are all sequential so that they fail to label a large volume of XML data in a timely manner. To address this issue, we devise parallel tree labeling algorithms for massive XML data. Specifically, we focus on how to efficiently label a single large XML file in parallel. We first propose parallel versions of two prominent tree labeling schemes based on the MapReduce framework. We then present techniques for runtime workload balancing and data repartition to solve performance issues caused by data skewness and MapReduce's inherited limitation. Through extensive experiments with synthetic and real-world datasets on 15 nodes, we show that our parallel labeling algorithms are up to 17 times faster than conventional algorithms, providing strong durability against data skewness.

Keywords Parallel computing · XML · Tree labeling algorithm · MapReduce

H. Choi · Y.-J. Lee
Department of Computer Science, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea

H. Choi
e-mail: hbchoi@dbserver.kaist.ac.kr

Y.-J. Lee
e-mail: yoonjoon.lee@kaist.ac.kr

K.-H. Lee (✉)
Intelligent Convergence Media Research Department, Broadcasting & Telecommunications Media Research Laboratory, ETRI, 218 Gajeong-ro, Yuseong-gu, Daejeon 305-700, Republic of Korea
e-mail: kyongha@etri.re.kr

Published online: 29 August 2013

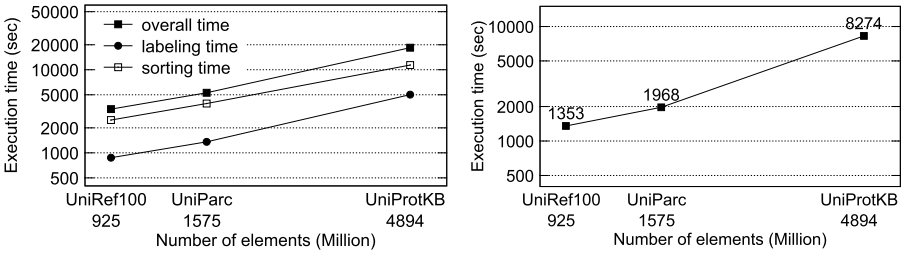
1 Introduction

XML is currently one of the most popular data formats for data representation and transmission on the Internet [9]. As many data have been typed in XML, the volume of a single XML document has also become enormous and also grows very quickly. For example, Wikipedia provides page dumps as a single XML document that sizes over 40 GBytes [4]. It is more often to witness large XML documents in scientific areas. For instance, UnitprotKB, which provides a collection of functional relationships between proteins, now hits more than 108 GBytes a file [5]. Moreover, the size of the XML document is continuously growing as biologists find new facts on the proteins in their experiments. Consequently, there is a growing demand for the support of query processing over a large XML document.

Meanwhile, labeling an XML document is the first and crucial step in XML query processing. Tree labeling algorithms facilitate XML query processing by assigning a unique label to each node in a tree that an XML document represents [19, 29, 30]. A structural relationship between two tree nodes is simply identified by comparing two labels that correspond to the nodes. Without tree labeling algorithms, XML query processing could be harder since there is no choice, but to traverse all nodes in an XML tree to find all occurrences of the subtree pattern that a given query represents.

However, conventional tree labeling algorithms are all sequential. This involves serious delays or halt in query processing as a system requires unbearable time to proceed its labeling process before actual query processing. Consequently, conventional algorithms can no longer label a massive XML document in a timely manner. To provide a rationale behind this assertion, we tested two popular tree labeling schemes, i.e., *interval-based* and *prefix-based* labeling schemes. We delicately implemented two labeling algorithms in Java. Figure 1 presents the result of this mini-test, which was performed on a Linux machine equipped with an AMD FX-4100 3.6 GHz processor, 8 GB memory, and a 7200 RPM HDD. In the test, all the labeling algorithms failed to label given XML files in a proper time. Specifically, the interval-based labeling algorithm spent approximately 5.56 hours (20,000 seconds) for labeling an XML document that consists of 4,894 million nodes. Note that we labeled both elements and attributes in the test.

To address this issue, we propose efficient parallel tree labeling algorithms for massive XML data in this article. Specifically, we focus on how to efficiently label a



(a) Interval-based labeling scheme (b) Prefix-based labeling scheme

Fig. 1 Execution time for sequential XML labeling algorithms

single large XML file in parallel. First, we provide MapReduce-based algorithms for two prominent tree labeling schemes. Then we present two optimization techniques for solving performance issues caused by data skewness and MapReduce's inherited limitations. Note that dynamic labeling techniques for XML document updates are out of scope in this article.

The key contributions of our approach are summarized as follows:

1. We provide an efficient method to parallelize conventional tree labeling algorithms with MapReduce. In other words, we provide an efficient way to tailor conventional tree labeling algorithms to fit the MapReduce programming model.
2. In our approach, elements are naturally grouped by tag name. In addition, elements with the same tag name are sorted in an ascending order of labels after the labeling process. This is just the same as the input type used in most of holistic twig join algorithms such as TwigStack [10]. Therefore, the results of our labeling algorithms are directly used for twig pattern joins with no more efforts.
3. In the MapReduce programming model, an input is partitioned into many equal-sized blocks with no semantic knowledge about its input data. This makes it difficult to label XML elements in parallel. We provide a method for keeping structural information of an XML document during the labeling process.
4. We also present optimization techniques to address performance issues in MapReduce. The skewed distribution of key-value pairs in input data may harm the overall performance of a MapReduce-based algorithm as the data skewness makes MapReduce tasks have imbalanced workloads. We address this performance issue by providing both a sophisticated runtime workload balancing and data repartition techniques.
5. We report results from a comprehensive set of experiments carried out to evaluate our MapReduce-based labeling algorithms in comparison with conventional tree labeling algorithms.

The rest of this article is organized as follows. Section 2 explains preliminary knowledge useful for better understanding of our work. Section 3 presents our parallel labeling techniques based on the MapReduce framework. Section 4 describes optimization techniques to alleviate a data skewness problem in labeling massive XML data. In Sect. 5, we extensively evaluate the performance of our parallel labeling algorithms with various datasets. Section 6 introduces previous work related to our approach. Finally, we conclude this article in Sect. 7.

2 Preliminaries

2.1 XML and Tree Labeling Schemes

An XML document is modeled as an *ordered*, *rooted*, and *labeled* tree [10, 19]. Each node in an XML document corresponds to an element or a value. Accordingly, each edge represents an element–element relationship or an element-value relationship. An XML element is represented by a pair of *start-tag* and *end-tag*. In addition, each element is identified by a *tag name* and an element may have attributes with their

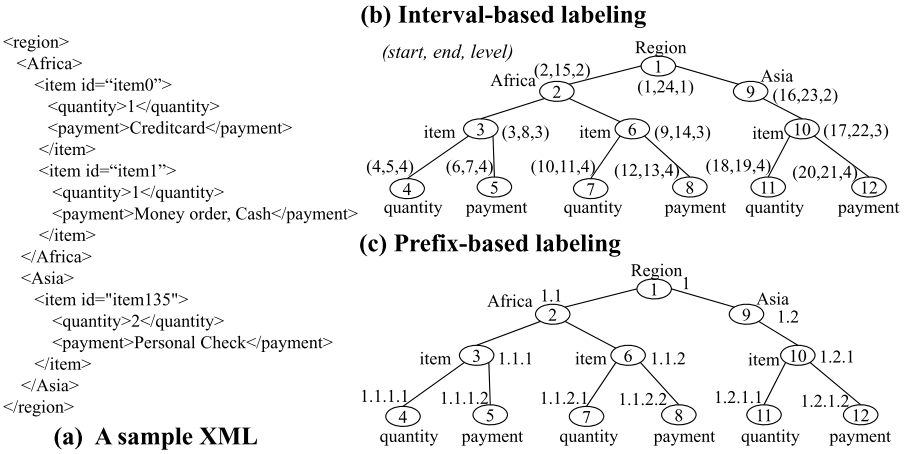


Fig. 2 XML and two tree labeling schemes

values. Figure 2(a) presents a sample XML document, which will be used throughout this article. For example, the `quantity` element is represented by a pair of a start-tag `<quantity>` and an end-tag `</quantity>` with value 2 in Fig. 2(a). An order of nodes in a tree that an XML document represents, aka *document order*, is obtained by a preorder traversal of the tree nodes.

XML query processing is considered as a subtree pattern matching problem which finds all occurrences of a given subtree pattern from a single large tree that an XML document represents. For example, the following XPath expression:

```
//item[quantity='1' and payment ="CreditCard"]
```

which matches all the `item` elements that (i) have a child element `quantity` with a value 1 and (ii) have a child element `payment` with a value "CreditCard." To process XML queries, relationships between elements such as ancestor–descendant and parent–child relationships must be identified. A tree labeling algorithm help users identify a relationship between two XML elements by simply comparing two labels associated with the elements. Users do not need to traverse whole nodes in an XML tree for query processing. Figure 2 (b) and (c) illustrate tree representations for the XML document shown in Fig. 2(a), which are labeled by two popular tree labeling schemes. Note that we do not illustrate any attributes and values in the trees for simplification.

2.1.1 Interval-Based Labeling Scheme

Interval-based labeling scheme or *region numbering* scheme helps identify a relationship between any two nodes in an XML tree with a 3-tuple defined as follows.

Definition 1 (Interval-based labeling scheme) Interval-based labeling scheme encodes the position of an XML tree node v as a 3-tuple (*start*, *end*, *level*), where a

pair of *start* and *end* indicates an interval of v and the *level* indicates the level of v in the XML tree.

For any two XML tree nodes u and v, u is an ancestor of v *iff* $u.start < v.start$ and $u.end > v.end$. In other words, u is an ancestor of v *iff* u's interval includes v's interval. A node u is a parent of a node v *iff* u is an ancestor of v and $v.level = u.level + 1$. Also, a node u precedes a node v in document order *iff* $u.start < v.start$. For example, in Fig. 2(b), the first `payment` element is a child of the first `item` element since the `item` element's interval [3, 8] includes the `payment` element's interval [6, 7] and level difference between two elements is 1. In the interval-based labeling scheme, a 3-tuple is completed when an end-tag occurs. Therefore, all labeled values are generated in postorder, different from a document order of an XML document. As a result, it is required to sort all the labels in document order.

2.1.2 Prefix-Based Labeling Scheme

In a prefix-based labeling scheme, a label for an XML tree node is a concatenation of its parent's label and its local order. We formally define the prefix-based labeling scheme as follows.

Definition 2 (Prefix-based labeling scheme) Prefix-based labeling scheme encodes the position of an XML tree node v, whose parent is u, by $L(v) := a_1.a_2.\cdots.a_m$ such that

1. $L(v)$ is a concatenation of $L(u)$ and v's local order, delimited by '.'.
2. The local order of v is i if v is the ith child of u.

For example, a label for the first `item` element in Fig. 2(c), "1.1.1," is a concatenation of its parent's label "1.1" and its local order "1." We call the sequence of components before local order, e.g., "1.1" in "1.1.1," a *prefix* of the label as it is inherited from its parent node's label. For any two nodes u and v, u is an ancestor of v *iff* the label of u is a prefix of the label of v. The u node is a parent of the v node *iff* the label of v has no prefix when removing the label of u from the left side of the label of v. For instance, a node labeled "1.1" is a parent of a node labeled "1.1.1," but not of a node labeled "1.2.1." Note that labels in the prefix-based labeling scheme are ordered by *Dewey order*. We define Dewey order, denoted by $\prec$, as shown below:

Definition 3 (Dewey order) Given two labels $L(u)$: $a_1.a_2.\cdots.a_m$ and $L(v)$: $b_1.b_2\cdots.b_n$, for any XML tree nodes u and v, $u \prec v$ *iff* either of the following two conditions holds:

1. $m < n$ and $a_1 = b_1, a_2 = b_2, \ldots, a_m = b_m$.
2. $\exists k \leq min(m, n)$ such that $a_1 = b_1, a_2 = b_2, \ldots, a_{k-1} = b_{k-1}$ and $a_k < b_k$.

The u node precedes the v node in document order *iff* $L(u) \prec L(v)$. For example, the `Africa` element precedes the third `item` element in document order since the label of the `Africa` element precedes the label of the third `item` element, i.e.,

1.1 $\prec$ 1.2.1. In prefix-based labeling scheme, level information is implicitly represented by the number of components in a label, e.g., an element labeled "1.2.1" is in level 3.

2.2 MapReduce

MapReduce is a parallel processing tool that provides ease-of-use, scalable, and fault-tolerant features [13]. MapReduce hides the details of the parallel execution so that users can focus only on their data processing strategies. The MapReduce programming model consists of `Map` and `Reduce` functions. The input for a MapReduce-based program is a list of (*key*1, *value*1) pairs and the `map()` function is applied to each key-value pair to compute intermediate results, i.e., (*key*2, *value*2) pairs. The intermediate key-value pairs are then grouped by *key*2 values, i.e., (*key*2, *list*(*value*2)). For each *key*2, the `reduce()` function reads a list of all values, *list*(*value*2), then produces an aggregated result.

We implement our algorithms with Hadoop [3], a well-known open-source implementation of MapReduce, since MapReduce itself is not available to the public for Google's proprietary use. Like MapReduce, Hadoop consists of two layers: a data storage layer called Hadoop DFS, aka HDFS, and a data processing layer called Hadoop MapReduce. HDFS is a block-structured distributed file system just like Google's GFS. A MapReduce-based program, referred to as a *job*, performs in two phases:the *map* and the *reduce* stages. Before starting the map stage, an input file is loaded on HDFS. At the loading time, the file is split into multiple fixed-size blocks. Hadoop MapReduce treats an input file as multiple InputSplits. An `InputSplit` is a chunk of an input file, which will be processed by a single map task. Each map task processes its InputSplit at a time. Note that an InputSplit can be more than one HDFS block if the size of InputSplit is chosen to be larger than HDFS block size.

A *mapper*, a worker that runs a map task, reads records, i.e., key-value pairs, from its InputSplit. Then the mapper applies the `map()` function to each record. The intermediate outputs produced by mappers are then sorted locally for grouping key-value pairs by key. After local sort, the `combine()` function is optionally applied to perform preaggregation to minimize the communication cost taken to transfer intermediate outputs to reducers. Then mapped outputs are stored on local disks of the mappers, partitioned into R partitions where R is the number of reduce tasks allowed in the MapReduce job. This is basically performed by *static hash-based partitioning scheme*, i.e., `hash(key) mod R`. When all map tasks are completed, the intermediate outputs are assigned to *reducers*, a worker that runs a reduce task. Basically, each record in the intermediate outputs is assigned to only a single reducer by *one-to-one shuffling* strategy. A reducer reads the intermediate results and merge them on the basis of *key*2, thus all values associated with a single *key*2 are grouped together. This grouping is internally done by *external merge-sort*. After that, each reducer applies the `reduce()` function to the grouped intermediate values for each *key*2. Finally, the outputs of reducers are stored on HDFS.

3 Parallel XML Labeling with MapReduce

Figure 3 shows the overall procedure of our parallel labeling techniques based on MapReduce. To begin with, an input XML document is loaded on HDFS. At the loading time, a large XML document is split into many HDFS blocks. Next, each map task labels XML elements in an InputSplit. By processing a file in chunks, we make multiple map tasks operate on a single file in parallel. However, map tasks cannot label all XML elements completely for two reasons. First, each map task does not have any knowledge about XML elements located in the other InputSplits since each map task works only on its InputSplit. Second, a start-tag and its corresponding end-tag may not be placed together in an InputSplit. For the reasons, each map task outputs labels which are partially completed. We call this process *partial labeling*. After the map stage, mapped outputs are shuffled by tag name and then sent to reducers. Each reducer then has the partial labels for all XML elements, which are associated with a tag name. At the reduce stage, each reducer merges the partial labels into a set of complete labels. We call this process *label completion*. To build a set of complete labels from partial labels, we exploit additional information called *offsets*, which are collected from every map task at the end of the map stage. The details of the label completion are deferred to Sects. 3.2 and 3.3.

3.1 Splitting XML Data with XMLInputFormat

The `InputFormat` class in Hadoop decides how to split an input file into multiple `InputSplits` and how to read key-value pairs from each InputSplit. The basic InputFormat class `TextInputFormat`interprets each line in an InputSplit 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. However, the `TextInputFormat` class is not suitable for processing XML documents due to its lack of XML semantics. It splits an input file only on the basis of InputSplit size. This causes improper splits of an XML document. For example, suppose that there is an XML document, which includes text `<author>Lee</author>` and the document is split into two InputSplit S_i and S_{i+1}, which hold "`<author>Lee</`" and "`author>`," respectively. While reading the InputSplit S_{i+1}, the incomplete end-tag `author>` raises a fatal parsing error

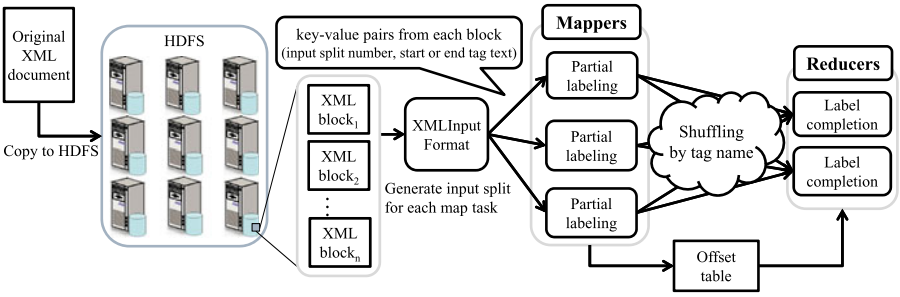


Fig. 3 Overview of parallel XML labeling with MapReduce

since it violates the XML grammar that an end-tag of an element must be in the form of `</'tagname'>`.

To address the issue, we write a custom InputFormat class, `XMLInputFormat`. The `XMLInputFormat` adjusts the boundary of each InputSplit such that all tags are not separated into two InputSplits. It checks the boundary of each InputSplit if any tag will be separated. If so, it extends the boundary of the InputSplit to include the tag. Furthermore, the `XMLInputFormat`class also interprets the records in an InputSplit on the basis of tags, unlike line-by-line key-value interpretation in the conventional `InputFormat` class. with the `XMLInputFormat` class, a key in each key-value pair is an identifier for the InputSplit, which contains a tag and a value is the tag itself.

Example 1 With the `XMLInputFormat` class, an XML document shown in Fig. 2(a) is interpreted as a series of key-value pairs, (1, ⟨`region`⟩), (1, ⟨`Africa`⟩), (1, ⟨`item id="item0"`⟩), (1, ⟨`quantity`⟩), (1, ⟨`/quantity`⟩) and so on.

3.2 Parallel Interval-Based Labeling Algorithm

Our parallel interval-based labeling algorithm works in two phases: *partial labeling* at the map stage and *label completion* at the reduce stage. Algorithm 1 describes the partial labeling process at the map stage.

In the algorithm, inputs are a list of key-value pairs, each of which is in the form of <InputSplitId, a tag>. At initialization, two variables *currentCount* and *currentLevel* are set to zero (line 2). For each tag, *currentCount* increases by 1 (line 4). If a tag is a start-tag, the algorithm build a new label with a *start* value which is set to the *currentCount* value and then the label is pushed into a stack *S* (lines 5–7). If a tag is an end-tag, the `map()` function pops a label from *S*, set an *end* value in the label to *currentCount*, and then emits the label as an output (lines 13–17). The *currentLevel* value is used to describe the level of an element. When a start-tag comes, the *currentLevel* increases by 1 and it is pushed into *S* together with the *start* value (lines 6–7). On the contrary, it decreases by 1 when an end-tag comes (lines 6 and 9).

The algorithm works similar to the conventional interval-based labeling scheme, but different in that it allows open-started and open-ended labels. An *open-started label* is a label that the *start* value of the label is still unknown, e.g., (x, 4, 2). An open-started label is encoded when the algorithm meets an end-tag, which is not paired by a corresponding start-tag in the same InputSplit. This happens if a start-tag and its end-tag are separately located in two different InputSplits. Similarly, an *open-ended label* is a label that the *end* value of the label is unknown yet, e.g., (10, x, 3). An open-ended label is encoded when the algorithm meets a start-tag, which does not have a corresponding end-tag in the same InputSplit.

If the algorithm meets an end-tag, which does not have its start-tag in the same InputSplit, it tries to pop a label from *S* even though *S* is empty. In the case, we make the `map()` function emit an open-started label (lines 10–11). On the contrary, if there are start-tags, which do not have corresponding end-tags in the same InputSplit, some open-ended labels are still left in *S* at the end of the map stage. Therefore, we emit all the open-ended labels from *S* in the end (lines 19–21). Note that

Algorithm 1 Partial labeling at the map stage

```
1:  Function INITIALIZE()
2:  initialize a stack S and set variables Count and Level to 0
3:  Function MAP(Integer inputSplitId, Text tag)
4:  increment Count by 1
5:  if tag is a start-tag then
6:      increment Level by 1
7:      build a new label L(Count, 0, Level)
8:      push L into S
9:  else                                                    ▷ if an end-tag
10:     decrement Level by 1
11:     if S is empty then
12:         build a new label L(0, Count, Level)            ▷ for an open-started label
13:     else
14:         L ← pop from S
15:         L.end ← Count
16:     end if
17:     emit(K, L)                          ▷ K := < tagname, inputSplitId, L.start >
18: end if
19: Function CLEANUP()
20: while S is not empty do                                 ▷ for open-ended labels
21:     L ← pop from S
22:     emit(K, L)
23: end while
24: write offsetInfo(InputSplitId, Count, Level) into HDFS
```

the *currentLevel* value can 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 mapped outputs is a composite key that is composed of the tag name, InputSplit Id, and the *start* value. This composite key scheme is useful for both grouping and sorting mapped outputs. With the key scheme, mapped outputs are grouped by distinct tag name first, then records in each group are sorted by InputSplit Id and the *start* value. Note that this grouping and sorting are naturally done by MapReduce's *merge-sort* based grouping strategy [13, 18]. Finally, each map task writes the final state of the *currentCount* and the *currentLevel* values on HDFS (line 23).

Figure 4 illustrates a comprehensive example of our parallel interval-based labeling algorithm. First, an input XML document is split into three InputSplits. Each map task reads records from its InputSplit with the `XMLInputFormat` class and then outputs partial labels. In the figure, the first map task reads tags located in the first InputSplit and then encodes partial labels `<4,5,4>`, `<6,7,4>`, `<3,8,3>`, `<2,x,2>`, and `<1,x,1>` where `<2,x,2>` and `<1,x,1>` are open-ended labels since start-tags of `Africa` and `region` elements are not paired by their end-tags in the InputSplit. Similarly, the second map task reads tags located in the second InputSplit and then generates partial labels. Note that partial labels have not been

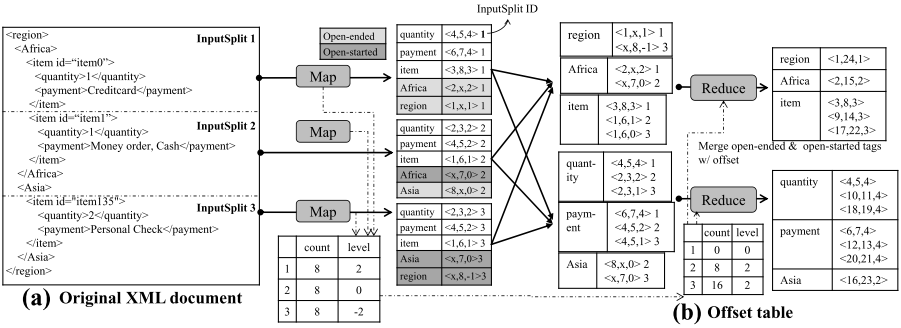


Fig. 4 An example of our parallel interval-based labeling algorithm

sorted in document order yet. Among the partial labels that the second map task encodes, `<x,7,0>` for the `Africa` element is an open-started label. Finally, each map task stores offset information *OffsetInfo*, which consists of the final state of the *Count* and the *Level* values with its InputSplit Id, on HDFS. In the figure, the first map task writes a tuple `(1, 8, 2)`, which represents that the first InputSplit has 8 tags including 2 open-ended tags. Again, the third map task writes a tuple `(3, 8, -2)` on HDFS, which represents that the third InputSplit has 8 tags including 2 open-started tags.

Algorithm 2 describes how we complete this labeling process with the partial labels. First, it builds an *offset table*, which provides the information that is required to complete partial labels at the reduce stage, with the offset information that map tasks have written on HDFS (line 2). An offset table consists of 3 columns, i.e., InputSplit Id, a cumulative *count* value, and a cumulative *level* value. In an offset table T, each row denoted by T_i is computed by

$$\begin{aligned} \mathrm{T}_i.\mathrm{count} &= \begin{cases} 0 & \text{if } i = 1 \\ \sum_{k=1}^{i-1} \mathrm{OffsetInfo}_k.\mathrm{count} & \text{otherwise} \end{cases} \\ \mathrm{T}_i.\mathrm{level} &= \begin{cases} 0 & \text{if } i = 1 \\ \sum_{k=1}^{i-1} \mathrm{OffsetInfo}_k.\mathrm{level} & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

Example 2 In Fig. 4, an offset table T has 3 rows as there are 3 InputSplits. The first row in T, denoted by T_1, has 0 count and 0 level as defined by formula (1). On the contrary, T_3's count and level is set to 16 and 2, respectively.

Next, the algorithm initializes a stack S, which is used for pairing open-ended labels to their corresponding open-started labels (line 3). Note that at the reduce stage, input data are grouped by tag name and also sorted in ascending order of (*inputSplitId*, *start*). The order is determined by the composite key scheme, i.e., (*tagname*, *InputSplitId*, *start*) as we described earlier. As a result, each `reduce()` function works with sorted label lists each of which represents the elements that have the same tag name.

Algorithm 2 Label completion at the reduce stage

```
   // Reducer class
 1: Function INITIALIZE()
 2: read offsetInfo from HDFS and build offset table T
 3: initialize stack S
 4: Function REDUCE(Key, label)
 5: i ← Key.InputSplitId
 6: if label L is open-ended then                    ▷ T_i is the i-th tuple in T
 7:     L.start ← L.start + T_i.count; L.level ← L.level + T_i.level
 8:     push L into S
 9: else if L is open-started then
10:     L.end ← L.end + T_i.count;
11:     pop L' from S
12:     L ← L ⊕ L'                          ▷ merge open-ended and open-started labels
13:     Output L as a final result
14: else
15:     L.start ← L.start + T_i.count;
16:     L.end ← L.end + T_i.count; L.level ← L.level + T_i.level
17: end if
```

There are three cases that the `reduce()` function works. First, when `Reduce()` function meets a label L which is neither open-ended nor open started, it simply adds $T_i.count$ to both $L.start$ and $L.end$. Also, it adds $T_i.level$ to $L.level$ (lines 14–15). Note that the offset table T keeps the cumulative tag-count and the level values for each InputSplit. Therefore, by simply adding the values in T_i to labels for the elements in the ith InputSplit, the labels are completed just as if all XML elements are numbered in serial from the beginning. Second, when the `reduce()` function meets an open-ended label L, it adds $T_i.start$ and $T_i.level$ to $L.start$ and $L.level$, respectively. Then it pushes L into S (lines 6–8). Last, when the `reduce()` function meets an open-started label L, it adds $T_i.end$ and $T_i.level$ to $L.end$ and L.$level$, respectively. Finally, it pops an open-ended label L' from S and then merges L' and an open-started label L into a single label L by a label-merge operator $\oplus$ defined as below (lines 9–12).

Definition 4 (Label-merge operator) Given an open-started label L and an open-ended label L', label-merge operator, denoted by $\oplus$, is defined by

$$L\langle x, end, level_1\rangle \oplus L'\langle start, x, level_2\rangle \rightarrow L\langle start, end, level_2\rangle$$

Example 3 In Fig. 4, the first reducer reads an open-ended label $L :< 2, x, 2 >$ for the `Africa` element in the first InputSplit. Then it adds values in T_1 to L and pushes L into S. No change occurs in L since values in T_1 are all zero. Next, the reducer now reads an open-started label $L' :< x, 7, 0 >$ from the second InputSplit, then it adds values in T_2 to L', i.e. $L' :< x, 7 + 8, 0 + 2 >$. Finally, the reducer pops L from S and then $L :< 2, x, 2 >$ and $L' :< x, 15, 2 >$ are merged into a single label $L :< 2, 15, 2 >$.

Note that since each reducer works with the sorted label lists grouped by tag name, each reducer always keeps open-ended tags in S in ascending order of start values for a single tag name. Therefore, whenever we pop an open-ended tag from S, the current open-started tag pairs with its corresponding open-ended tag. It is always satisfied even if elements are recursive, e.g., `<Africa><Africa>...</Africa></Africa>`.

3.3 Parallel Prefix-Based Labeling Algorithm

As described in Sect. 2.1.2, a label for an XML element is a concatenation of its parent's label and its local order in the prefix-based labeling scheme. Figure 5 illustrates a comprehensive example of our parallel prefix-based labeling algorithm.

First, each map task reads key-value pairs from its InputSplit. A map task works with two variables, a vector *label* that keeps the label for one's parent and a numeric variable *localorder* that keeps one's local order. The *label* and the *localorder* are initially set to empty and 0. Whenever a map task reads a start-tag, a mapper generates a new label by concatenating *label* and *localorder*$+1$ unless the *label* vector is empty. Then it puts the new label into the *label* vector and set the *localorder* value to 0 again. For the first start-tag in an InputSplit, a label is just *localorder* $+ 1$ since the *label* vector is empty. Whenever a map task meets an end-tag, it removes a postfix from the *label* vector and replace the *localorder* value with the postfix unless the *label* vector is empty. When the *label* vector is empty, the *label* vector stays empty and the *localorder* value is reset to 0 (This is for the same case of the open-started tag as in the previous Sect. 3.2). In this way, we label all elements in parallel.

However, the labels are still incomplete since each map task works only with its InputSplit. For example, the `item` element in the second InputSplit is labeled as 1 at Map stage in Fig. 5. A final label for the element must be 1.1.2 since it is the second child of the `Africa` element which is labeled as 1.1. We complete partial labels at the reduce stage. This process is called *label calibration*. To do this, we use map tasks' final states of *label* and *localorder*. For example, we calibrate the partial label 1 with the final states of *label* and *localorder* of the first map task, i.e., 1.1 and 1. The details are described later in this section.

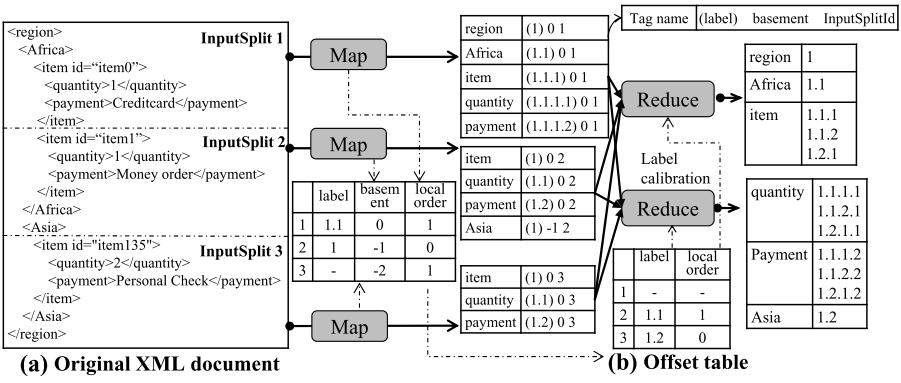


Fig. 5 An illustration of parallel prefix-based labeling

Nonetheless, the information is still insufficient to restore a global Dewey order. For example, both `item` and `Asia` elements in the second InputSplit are labeled as 1 in Fig. 5. This happens since the `Asia` element is the first element in level 0 while the `item` element is in level 1. It is not surprising that some elements are under level 1 since some elements remain unpaired in an InputSplit after the map phase in our algorithm.

To complement this, we use an additional variable *basement*, which decides how many postfixes should be removed from *label*. The *basement* value for an XML element indicates how many end-tags, which are not paired by their start-tags exist before the element within an InputSplit. If the *basement* value for an XML element is 0, no end-tag which is not paired by its start-tag exists before the element in an InputSplit. On the contrary, the *basement* value for the `Asia` element is set to -1 in the figure since an end-tag `</Africa>`, which is not paired by a start-tag, appears before the element in the second InputSplit.

Suppose that we have a basement value -1 and a partial (incomplete) label X. When we calibrate the partial label with a given prefix 1.1.2, we remove the last postfix from the prefix by the basement value -1, i.e., 1.1.

Map tasks output partial labels in forms of $<label, basement, inputSplitId>$. Moreover, each map task writes offset information as a 3-tuple $<label, basement, localorder>$ on HDFS before it ends. The offset information is collected to build an offset table T similar to the algorithm in the previous section.

We now define the *label-calibrate* operator, which is used for both building an offset table and label calibration.

Definition 5 (Label-calibrate operator) Given two tuples $X : \langle x_1.x_2.\ldots.x_m,\ b_x,\ l_x\rangle$ and $Y : \langle y_1.y_2.\ldots.y_n,\ b_y,\ l_y\rangle$, label-calibrate operator, denoted by $\odot$, is defined by

$$X \odot Y \equiv \begin{cases} \langle x_1.x_2.\ldots.x_m.(y_1 + l_x).y_2.\ldots.y_n,\ 0,\ l_y\rangle & \text{if } b_y = 0 \\ \langle x_1.x_2.\ldots.x_{m+b_y},\ 0,\ x_{m+b_y+1} + l_y\rangle & \text{if } b_y \neq 0, n = 0 \\ \langle x_1.x_2.\ldots.x_{m+b_y}.(x_{n+b_y+1} + y_1).y_2.\ldots.y_n,\ 0,\ l_y\rangle & \text{if } b_y \neq 0, n > 0 \end{cases} \tag{2}$$

where x_i and y_i represent the ith items of two labels. The b and the l represent the *basement* value and the *localorder* value, respectively.

The label-calibrate operator calibrates a label for an XML element e with the tuple for e's preceding element. We explain how it works in each case with examples.

Example 4 Suppose that there are two tuples X and Y and X is set to $<1.1, 0, 2>$.

1. **The 1st case** For a tuple Y: $<1.1, 0, 1>$, the result of label calibration is $<1.1.3.1,0,1>$ since $X \odot Y = <1.1, 0, 2> \odot <1.1, 0, 1> = <1.1.(2+1).1, 0, 1>$ when $b_y = 0$.
2. **The 2nd case** For a tuple Y: $<empty, -1, 1>$, $X \odot Y = <1, 0, 2>$ since $b_y \neq 0$, $n = 0$.
3. **The 3rd case** For a tuple Y: $<1.1, -1, 1>$, $X \odot Y = <1.(1 + 1).1, 0, 1>$ since $b_y \neq 0, n > 0$.

Note that the operation $\odot$ is neither commutative nor associative. We now define an offset table used in our parallel prefix-based labeling algorithm.

Definition 6 (Offset table) An offset table T consists of tuples in forms of $<label, basement, localorder>$ and the ith tuple T_i of T is computed as follows:

$$\mathrm{T}_i = \begin{cases} \langle empty,\ 0,\ 0\rangle & \text{if } i = 1 \\ \bigodot_{k=1}^{i-1} \mathrm{OffsetInfo}_k & \text{otherwise} \end{cases} \tag{3}$$

where $\bigodot$ is a cumulative operator that is defined by $\bigodot_{i=1}^{n} x_i \equiv x_1 \odot x_2 \odot \cdots \odot x_n$.

The offset table T is similar to the offset table in Sect. 3.2 except column types that it holds. In real application, we can omit *basement* column in the offset table since the column values are always zero in T. However, *basement* values in mapped outputs are still kept for label calibration.

Finally, we define label calibration which computes global labels by calibrating partial labels with the offset table T.

Definition 7 (Label calibration) Given a label L for an XML element in the ith InputSplit with basement b, label calibration builds a complete label L' as follows:

$$T' = T_i \odot < L, b, \_ > \tag{4}$$

where T_i is the ith tuple of an offset table T and L' is the label in the tuple T'.

Example 5 In Fig. 5, the `item` element in the second InputSplit is labeled as 1 with 0 basement at Map stage. Thus, a mapped output for the element is recorded as <1, $0, \_ >$ (local order is not involved in the record). Since the element is in the second InputSplit, T_2 is referred to calibrate the label for the element. That is,

$$T' = T_2 <1.1, 0, 1> \odot < 1, 0, \_ > = < 1.1.(1+1), 0, \_> \therefore L' = 1.1.2.$$

Example 6 In Fig. 5, the `Asia` element in the second InputSplit is labeled as $<1, -1, \_>$. The complete label for this element is computed by

$$T' = T_2 <1.1, 0, 1> \odot < 1, -1, \_> = < 1.(1+1), 0, \_ > . \therefore L' = 1.2.$$

In our implementation, we compress all the labels in DLN(Dynamic Level Numbering) scheme introduced in [8] for compact label representation, which results in reducing both I/O cost and space overhead during the labeling process.

4 Optimizations

As described in the previous section, we use distinct tag names as keys in our algorithms. This enables to group all XML elements by tag name. All XML elements with the same tag name flock to a reducer by virtue of the hash-based partitioning strategy

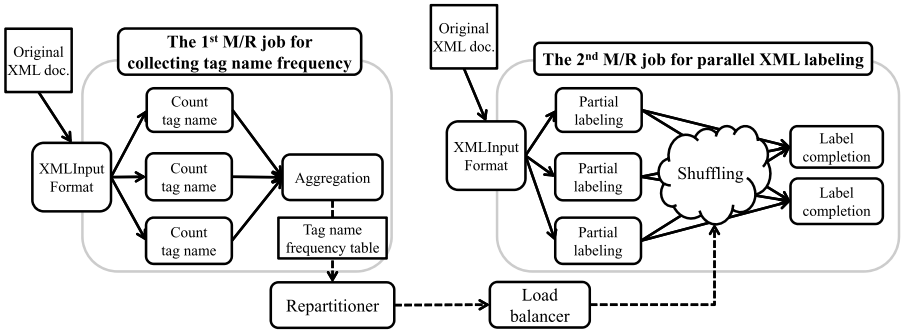


Fig. 6 Overview of the optimized version of parallel labeling process

in the MapReduce framework. However, the hash-based partitioning strategy may skew workloads assigned across reducers. If element frequencies are skewed in an input XML, workloads of MapReduce tasks are also apt to be skewed. Suppose that we process an XML document, which consists of 3 distinct tag names `A`, `B`, and `C`. We further suppose that the element frequency for each tag name is 150, 50, and 50, respectively. If the number of reducers in the MapReduce program is 2, the first reducer is assigned to `A` and `C` elements, 200 elements in total, while the second reducer has only 50 `B` elements in the hash-based partitioning strategy. This workload imbalance makes MapReduce-based algorithms inefficient. It becomes worse at the reduce stage since reducers are not balanced well by runtime scheduling as much as mappers do [17]. Therefore we focus on balancing workloads across the reducers.

Furthermore, we observe that key cardinality is sometimes much less than the number of reducers. It causes another problem that some reducers do not work at all as any elements are not assigned to them. This significantly limits the level of parallelism in our MapReduce-based algorithms. To address the issues, we suggest workload balancing and data repartition techniques. Figure 6 overviews the overall procedure of our parallel labeling algorithm, which includes the optimization techniques that we describe in this section.

Our optimization techniques require the information of element frequencies. To achieve this, we apply an additional MapReduce job before the MapReduce job for the parallel labeling process. The additional job simply counts the element frequency for each distinct tag name and then records the statistics on HDFS. After that, our optimization module optimizes parallel labeling process by repartitioning inputs and balancing workloads at the reduce stage according to the statistics. It is undoubtable that the additional MapReduce job imposes an extra cost to the parallel labeling process. However, we claim with our experimental results that the cost of the additional MapReduce job are well compensated by the improved overall performance of the labeling process.

4.1 Runtime Workload Balancing

We first define our workload balancing problem as follows:

Definition 8 (Workload balancing problem) Given a set S of n distinct tag names coupled with element frequency $e_1, e_2, \ldots, e_n$, we partition S into R subsets, $S_1, S_2, \ldots, S_R$ such that the sum of element frequencies in each subset is equal. The subsets $S_1, S_2, \ldots, S_R$ are disjoint and they cover S.

This problem can be reduced to an optimization version of a well-known NP-complete decision problem, *k-partition problem* [12]. Since there is no pseudo-polynomial solution known in literature, we solve this with a heuristic approach based on a greedy approximation algorithm, known as *first-fit decreasing algorithm* in literature [6]. Algorithm 3 describes our workload balancing algorithm. Before running the algorithm, we first sort the list of distinct tag names in descending order of element frequency, which are acquired by the first MapReduce job in our parallel labeling process. Note that since the number of distinct tag names is small, sorting cost can be ignorable (refer to Table 2). With a sorted list of element frequencies *FreqList*, the algorithm first assigns R tag names whose element frequencies are the highest in the list to R reducers such that each reducer has a single tag name. The algorithm then computes the cost of each reducer with the element frequency of the tag name assigned (lines 2–5). After that, the algorithm selects a reducer whose cost is the lowest and then assigns another tag name whose element frequency is the highest in the list to the reducer. It iterates until all tag names in *FreqList* are assigned (lines 7–11). Note that whenever tag names are assigned, the tag names are removed from *FreqList*. Finally, the algorithm returns the *allocationMap* data structure that describes, which reducer a key-value pair is assigned to. We update the default partitioning strategy in Hadoop in a way that it guides each key-value pair to a suitable reducer according to the *allocationMap*.

It is straightforward that the time complexity of this algorithm is $O(R + N \cdot \log R)$ where N is the number of distinct tag names. In our application, both N and R are small even in real applications so that this algorithm can be simply run in memory.

Algorithm 3 Workload balancer

FreqList := a list of (*tagname*, *frequency*) sorted by *frequency*
cost[R] ← all 0 ▷ R : the number of reducers
1: **Function LoadBalancer**(*keyreqList*)
2: **for** *reduceID* = 1 → R **do**
3: selects a tag name k whose element frequency is the highest from *FreqList*
4: put a pair (k, *reducerID*) into *allocationMap*
5: add *frequency* of k to *cost*[*reduceID*]
6: **end for**
7: **for each** tag name k in *FreqList* **do**
8: find the *reduceID* of a reducer with the smallest cost
9: put a pair (k, *reducerID*) into *allocationMap*
10: add *frequency* of k to *cost*[*reduceID*]
11: **end for**
12: **return** *allocationMap*

4.2 Data Repartition

Sometimes, only a few distinct tag names appear in an XML document. In the case, the level of parallelism is restricted despite the workload balancing algorithm in Sect. 4.1. For example, only three tag names occupy more than 75 % of element population in UniRef100 XML document as shown in Fig. 8. In the case, some computing nodes fail to get assigned XML elements in a balanced way. For that reason, we cannot fully utilize all nodes in a cluster if a few tag names exhibit high occupancy in an Input XML document. Moreover, this give rise to a performance problem in that only a few of reducers are assigned to too many XML elements. The heavy-loaded tasks defer the overall process since a MapReduce-based program will not complete its job until all of its tasks end. This problem becomes worse as the number of distinct tag names in an XML document is much less than the number of reducers.

We address this issue by repartitioning mapped outputs in a balanced way before the second Reduce stage. Our key idea is that given the total number of XML elements m and the number of reducers R, each reducer is guided to process at most m/R partial labels. To do this, we repartition a list of partial labels, which exceeds m/R into several lists. Meanwhile, we also keep the number of repartitions as few as possible in order to merge the labels from different repartitions. Note that the total number of XML elements and its mean value are counted by the first MapReduce job as shown in Fig. 6. Also note that XML elements which belong to a single tag name may come from multiple InputSplits. Moreover, the mapped outputs that we repartition here are partial labels which are not fully labeled. Therefore, we need to record which InputSplits the XML elements that correspond to the partial labels come from. The information is used to merge the results into a final list of fully-computed labels.

Algorithm 4 describes our repartitioning algorithm in detail. First, it finds all tag names of which element frequency exceeds *meanFreq* each (line 2). Next, it examines which InputSplits constitute an element list for the tag name k, denoted by *ListK*, with frequency distribution (line 3). Next, it partitions *ListK* into $ListK_1, ListK_2, \ldots, ListK_n$ such that $|ListK_i| \leq meanFreq, \forall i < n$. To achieve this, it checks element frequency information in each InputSplit (lines 5–6). When the sum of element frequencies for a tag name exceeds *meanFreq*, it partitions the element list by creating a new tag name whose element list is composed of the elements that appear in from the *startID*-th to the $(i-1)$-th InputSplit (lines 7–8). Next, it replaces the original tag name k in *FreqList* with the new partitioned keys (line 14). Finally, we return the *FreqList* to the workload balancer (lines 15–16). It is intuitive that the time complexity of this algorithm is simply $O(t \cdot n)$ where t is the number of distinct tag names and n is the number of the input splits.

Example 7 Suppose that we have 4 reducers and 4 distinct tag names `A`, `B`, `C`, and `D`. Table 1 describes element frequency information for each tag name and how the element lists are composed of XML elements from multiple InputSplits. The *meanFreq* value is 100 since $m/R = 400/4$. First, our repartitioning algorithm selects a list of `B` elements to partition since $|ListB|$ exceeds *meanFreq*. The `ListB` is partitioned into two partitions, $ListB_1$ and $ListB_2$ where $ListB_1$ consists of elements only from InputSplit 1 while $ListB_2$ consists of elements from InputSplit 2–4. Other element

Algorithm 4 Data repartitioner

FreqList := a list of (*tagname*, *frequency*) sorted by *frequency*
DistributionMap := a list of (*tagname*, *freqArray*[n]) ▷ n is the number of InputSplits
freqArray[*i*] := element frequency in the *i*-th InputSplit
meanFreq := the total number of elements in an input/ the number of reduces
1: **Function Repartitioner**(*keyFreqList*, *keyDistributionMap*, *meanFreq*)
2: **for all** tagname *k* in *FreqList* that satisfies *frequency* > *meanFreq* **do**
3: *FreqArray* ← *DistributionMap*.get(*k*)
4: *startID* ← 1, *partitionID* ← 1, *currentFreq* ← *FreqArray*[1]
5: **for** $i = 2 \rightarrow n$ **do**
6: **if** *currentFreq* + *FreqArray*[i] > *meanFreq* **then**
7: create new $k_{partitionID}$ whose elements appear in from *startID*-th to the $(i-1)$-th InputSplit
8: *partitionID*++, *currentFreq* = 0, *startID* = *i*
9: **else**
10: *currentFreq*+ = *FreqArray*[i]
11: **end if**
12: **end for**
13: create new $k_{partitionID}$ whose elements appear in from the *startID*-th to the *n*-th InputSplit
14: replace *k* in *FreqList* with $k_1, k_2, \ldots, k_{partitionID}$
15: **end for**
16: **return** updated *FreqList*

Table 1 An example of element frequencies

# of elements	Tag name A	B	C	D
InputSplit 1	20	100	10	10
InputSplit 2	30	50	10	10
InputSplit 3	20	30	20	10
InputSplit 4	30	20	10	20
Element list size	100	200	50	50

lists are not partitioned since their sizes are not bigger than *meanFreq*. Finally, Inputs for reducers in the second M/R job are set to *ListA*, $ListB_1$, $ListB_2$, *ListC*, and *ListD*.

5 Performance Study

5.1 Experimental Setup

We implemented our algorithms with Hanborq Distribution of Hadoop (HDH), which features a fast job launching and low-latent task transitions [24]. We ran all experiments on a cluster of 16 machines running on Ubuntu 10.10 unless stated otherwise.

Table 2 Statistics of XML dataset

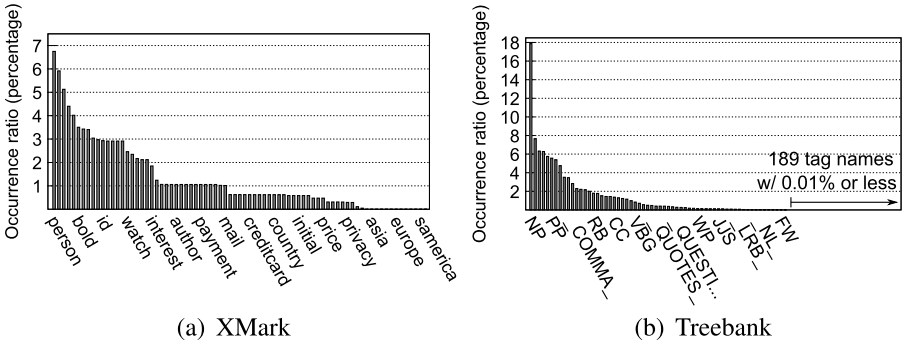
Filename	XMark1000	TreeBank1000	UniRef100	UniParc	UniProtKB
File size(KB)	117,159,962	84,064,928	25,088,663	38,334,953	108,283,066
# of elements	1,670,594,672	2,437,665,001	335,153,446	360,376,852	2,110,330,358
# of attributes	383,127,024	1	589,568,839	1,215,063,103	2,783,354,175
Depth in avg.	4.738	6.873	4.565	3.775	4.333
Max depth	12	36	6	5	7
# distinct paths	548	338,749	30	24	149
# distinct tag names	77	251	19	23	80

Table 3 Hadoop settings for our experiments

Parameter	Value
The number of nodes	16 (1 for master, 15 for slave nodes)
The number of mappers per node	8
The number of reducers per node	4
HDFS block size	64 MB (by default)
InputSplit size	64 MB (by default)
Replication factor	3 (by default)

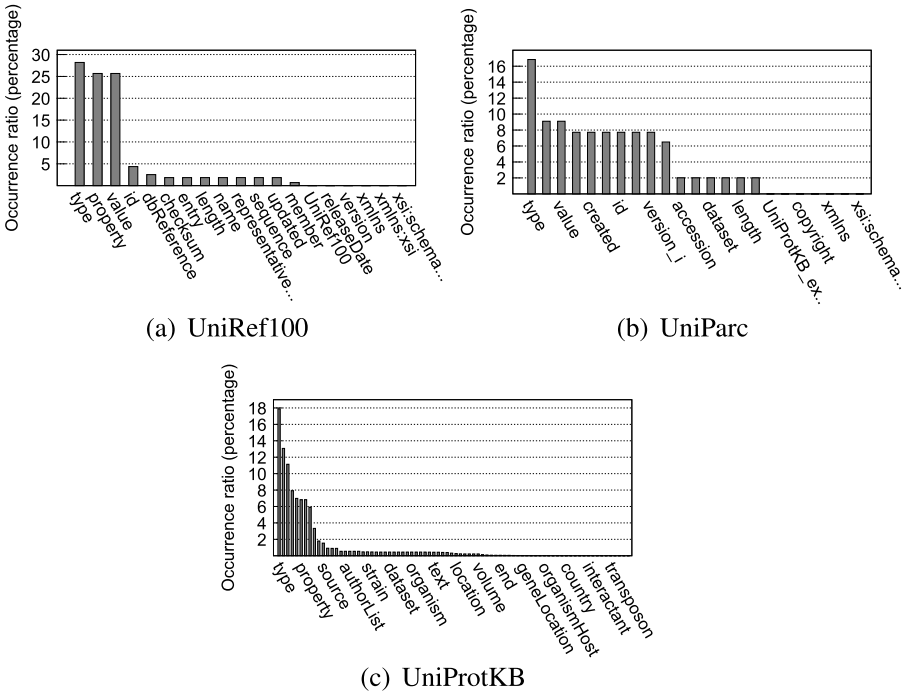
A node works as a master and the other nodes are designated as slaves. Each node is equipped with two Intel Xeon E5620 2.4 GHz CPU, 7200 RPM HDD, and 16 GB memory. All nodes are connected through a gigabyte switching hub. Table 3 presents the information about our Hadoop settings. This setting was used throughout our experiments. All other settings were set to default values for fair comparison.

We tested our parallel XML labeling algorithms with two synthetic and three real-world XML data [5, 25, 28], which have been widely used in many XML-related researches. Table 2 presents the statistics of XML datasets used in our experiments. We generated XMark dataset with scale factor 10, 100, and 1,000 [25] for synthetic XML data. We also generated synthetic XML documents based on Treebank XML [28] to test our labeling algorithms with deep-structured XML documents, which contain many distinct root-to-leaf paths. For the real-world dataset, we selected three real-world XML data, i.e., UniRef100, UniParc, and UniProtKB, which represent protein sequences and their functional information [5]. Figures 7 and 8 present the distribution of element frequencies in our XML dataset. Note that element frequency distribution is more skewed in Treebank than in XMark. In fact, 18 percentile of XML elements belong to a single tag name while other 189 tag names occupy less than 0.01 % of element population in Treebank. Furthermore, UniRef100 and UniParc exhibit not only skewed distribution of element frequencies but also a limited number of distinct tag names. More than 300 million elements belong to only 19 distinct tag names in UniRef100.



(a) XMark (b) Treebank

Fig. 7 Tag name frequency for the synthetic dataset



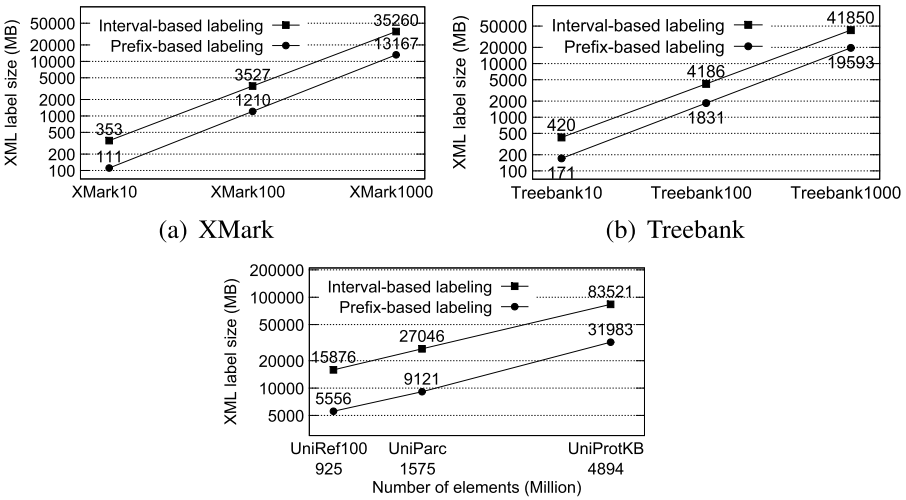
(a) UniRef100 (b) UniParc

(c) UniProtKB

Fig. 8 Tag name frequency for the real-world dataset

5.2 Performance Analysis

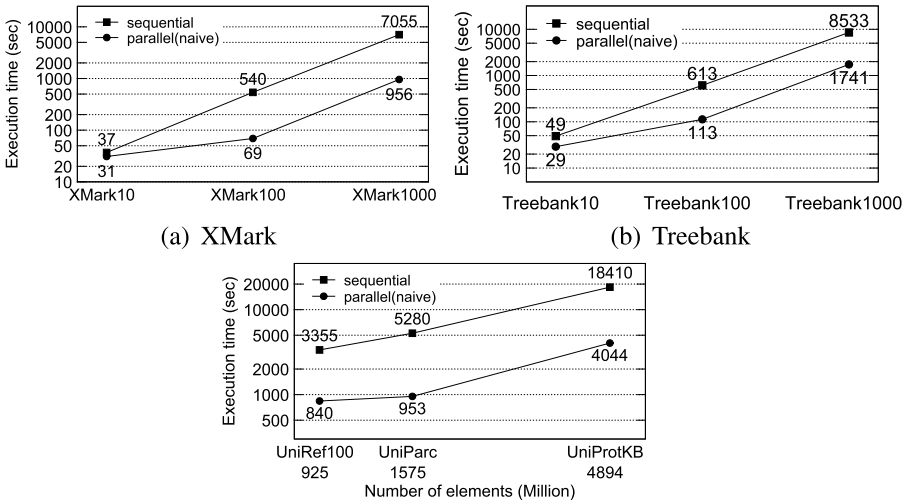
First, we compared two labeling schemes in terms of label size. Figure 9 presents the size information of two label types. As shown in the figure, the size of prefix-based labels is two or more times smaller than that of interval-based labels. The reason is that prefix-based labels are compressed by DLN scheme [8] in our experiments unlike interval-based labeling scheme. Note that the overall size of labels is always same whether the labels are computed in serial or in parallel.



(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 9 The size of XML labels



(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 10 Sequential vs. parallel algorithm for interval-based labeling

We then compared our parallel algorithms with conventional (sequential) algorithms. The results are shown in Figs. 10 and 11. Note that all the parallel labeling algorithms in the figures are naive now that they are not optimized by two techniques described in Sect. 4. The performance of our parallel algorithms did not outplay sequential algorithms when the size of an input XML document is small, e.g., XMark10 and TreeBank10. In the cases, some mappers did not work at all as they had not been assigned to any InputSplit. This happens when an input XML document

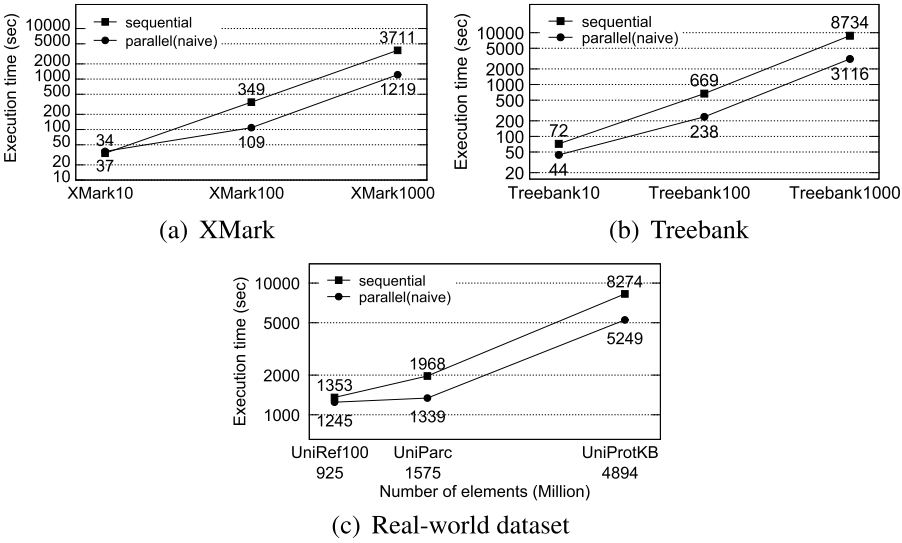


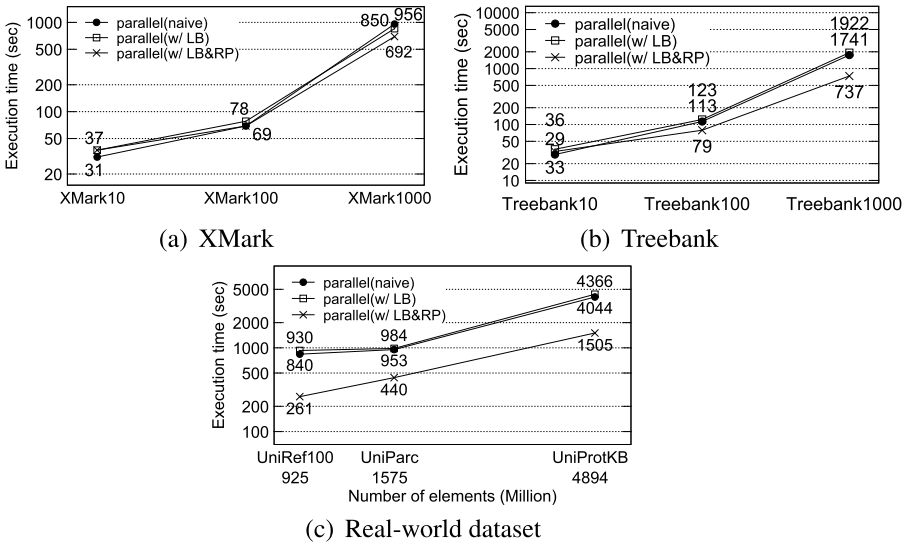
Fig. 11 Sequential vs. parallel algorithm for prefix-based labeling

is not big enough to assign its InputSplits to all the mappers. However, when an input XML document is big enough, our parallel algorithms clearly outperforms sequential labeling algorithms. Specifically, parallel interval-based labeling algorithm showed better speedup than parallel prefix-based labeling algorithm showed. The reason is that the interval-based labeling algorithm requires sorting, which is naturally applied during processing in MapReduce. However, when it comes to speedup and efficiency per node, these results of naive approaches are not impressive. The best speedup of our parallel interval-based labeling scheme achieved with 15 slave nodes recorded x3.8∼x7.3 as shown in Figs. 10 and 11. In other words, efficiency per node is 0.253∼0.486. It is a strong evidence that a naive MapReduce-based program does not offer remarkable performance in the parallel labeling process. We show the effectiveness of our optimization techniques in the following section.

5.3 Optimization

We compared the optimized versions of our parallel algorithms with the naive algorithm. Figure 12 shows the results of three versions of our parallel interval-based labeling algorithm: a naive algorithm that has no optimization (`naive`), a parallel algorithm which has the feature of workload balancing (`w/LB`), and an optimized algorithm with both workload balancing and data repartition (`w/LB&RP`). Figure 13 presents the results of three versions of our parallel prefix-based labeling algorithm.

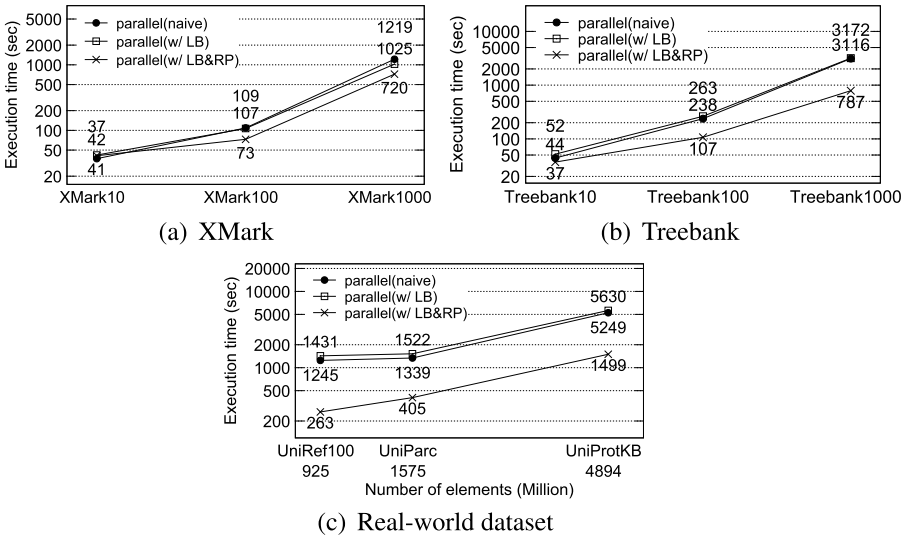
Note that the execution times in the figures are depicted in log scale. Our optimization techniques improved the performance of parallel labeling algorithms even though it required an additional MapReduce job. This result is more clear in cases of Treebank and real-world dataset. The reason is that the distribution of element frequencies is more skewed in the dataset, but also the number of distinct tag names



(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 12 The effect of optimizations in parallel interval-based labeling

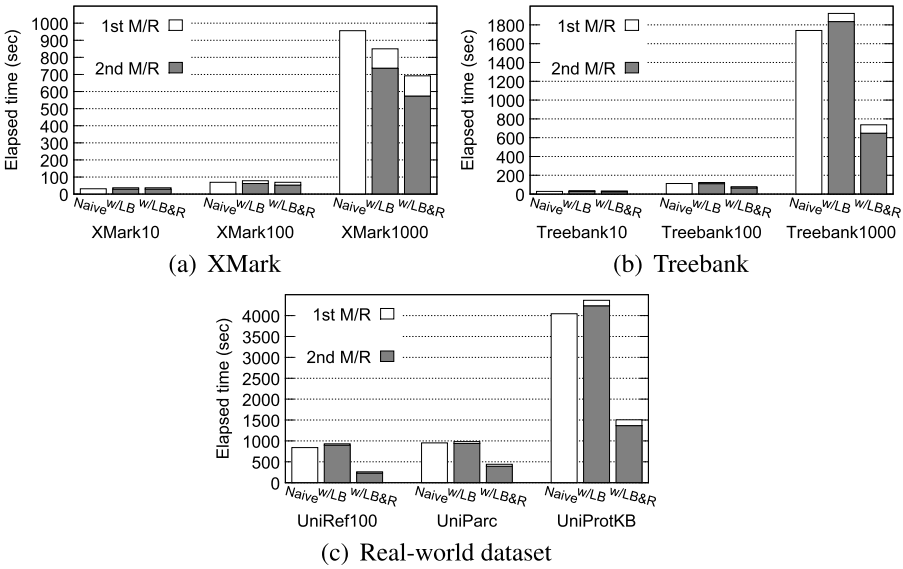


(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 13 The effect of optimizations in parallel prefix-base labeling

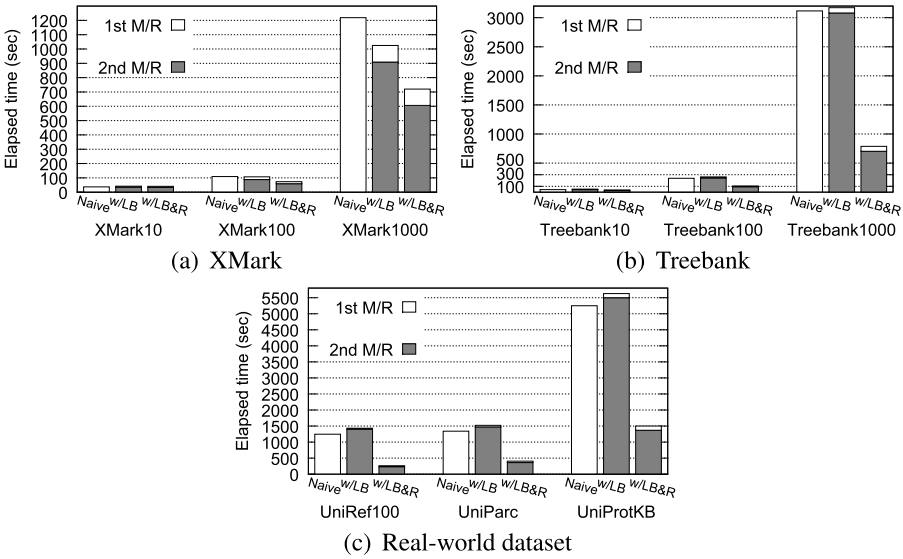
are much fewer. Therefore, there are more chances to be optimized by both workload balancing and data repartition in the dataset. The cumulative bar graphs in Figs. 14 and 15 present the execution time breakdown of three parallel labeling algorithms that processed synthetic and real-world dataset. If the volume of input XML data is not big enough, the performance gain is marginal since the substantial MapReduce job



(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 14 The execution time breakdown of parallel interval-based labeling algorithms



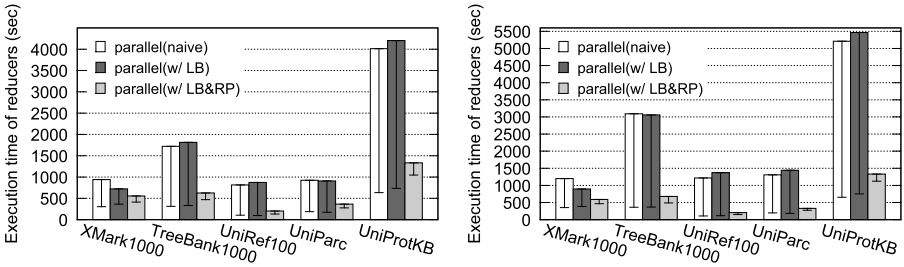
(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 15 The execution time breakdown of parallel prefix-based labeling algorithms

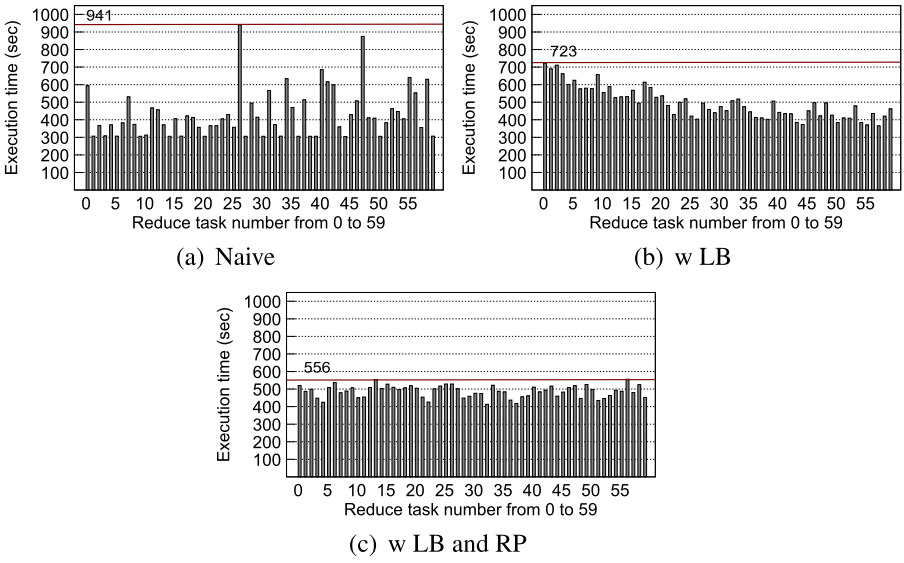
acts as a burden that the parallel labeling algorithm shoulders. However, the bigger the volume of input XML data, the better performance gain we could achieve.

Figure 16 shows how much the optimization techniques improved parallel labeling algorithms for a massive volume of XML data. In all cases, an algorithm optimized by both workload balancing and data repartition always showed the best performance.



(a) Interval-based labeling scheme (b) Prefix-based labeling scheme

Fig. 16 The effect of workload balancing and input repartition



(a) Naive (b) w LB

(c) w LB and RP

Fig. 17 The execution time at the reduce stage (XMark1000 in interval-based labeling scheme)

Note that a vertical line in each bar represents the difference between the shortest and the longest execution time of reducers in the second MapReduce job. A long vertical line implies that workloads have not been evenly distributed across reducers, involving the long overall execution time. On the contrary, a short vertical line implies that workloads have been evenly distributed across nodes. Note that the high end of a vertical line always meets with the overall execution time of the algorithm since a MapReduce job ends when all reducers end.

To further test the effect of workload balancing and data repartition, we also examined the execution time of all reducers in the second MapReduce job. The results are shown in Figs. 17 and 18. We observed that our optimization techniques effectively mitigated the data skewness problem by balancing workloads across reducers.

Finally, we investigated the scalability of our parallel labeling algorithms with the optimization features. The results are presented in Figs. 19 and 20. As shown in

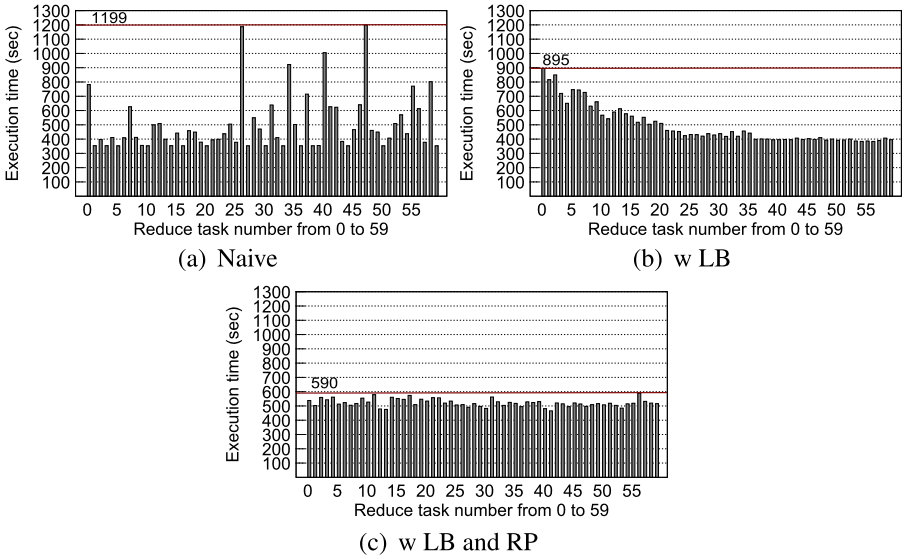


Fig. 18 The execution time at the reduce stage (XMark1000 in prefix-based labeling scheme)

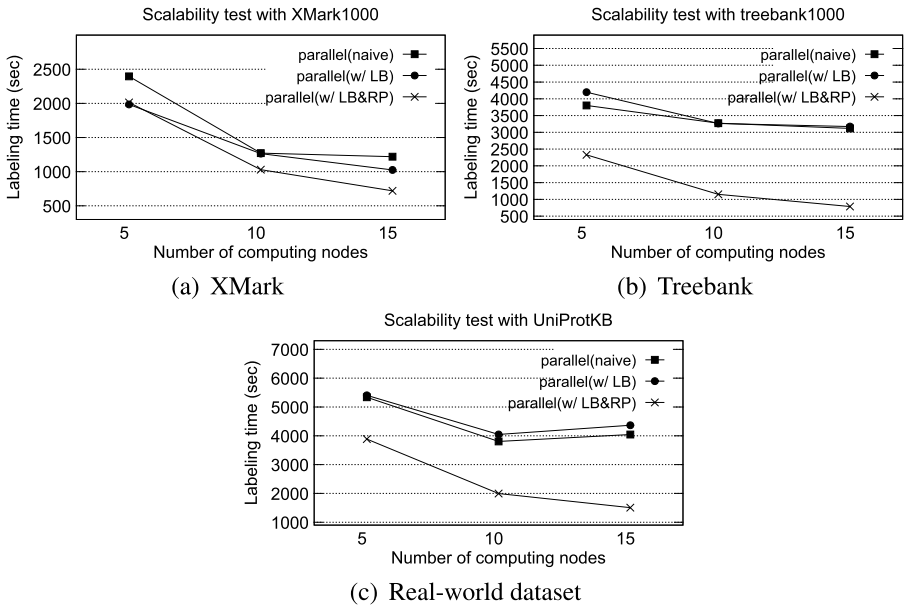
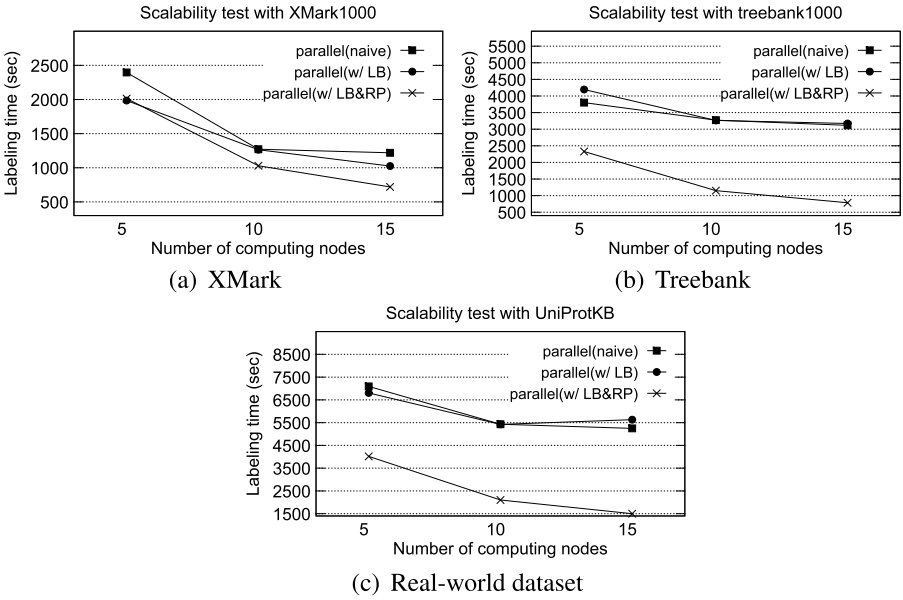


Fig. 19 Scalability of parallel interval-based labeling algorithms

the figures, naive algorithms revealed a scalability issue. Thus, corresponding graphs flattened as the number of nodes increased. On the other hand, algorithms with the features of workload balancing and data repartition scaled nicely.



(a) XMark (b) Treebank

(c) Real-world dataset

Fig. 20 Scalability of parallel prefix-based labeling algorithms

6 Related Work

It is known in literature that Dietz's number is the first work which provides a labeling scheme for trees [14]. He labeled each tree node with a pair of numeric values, *pre* and *post*, each of which represents preorder and postorder traversal numbers of the tree node, respectively. Zhang et al. adapted Dietz's number into XML query processing in a relational database [30]. They labeled an XML element with a 3-tuple of (*start*, *end*, *level*), each of which describes the position of the start-tag, the position of the end-tag, and level of the XML element, respectively. This approach is also known as *region* or *interval-based numbering scheme* in literature. Li et al. extended Dietz's number to avoid frequent relabeling of whole XML elements when new elements are inserted [19]. BaseX, an open-sourced XML DBMS [1], also uses a modified version of Li's approach in processing XML queries [15].

Prefix-based labeling schemes originated from Dewey decimal classification, which was invented for library classification by M. Dewey [26]. Tatarinov et al. first introduced the prefix-based labeling scheme called *Dewey ID* into XML query processing [29]. In their approach, each XML element is associated with a vector of identifiers which represents a path from the root to the element. An advantage of the prefix-based labeling scheme over the interval-based labeling scheme is an ability to avoid relabeling whole XML elements when updates on the XML tree occurred. Due to the easy maintenance under dynamic updates on XML trees, some commercial DBMS have used prefix-based labeling schemes in XML query processing [7, 21, 22]. ORDPATH is a novel prefix-based labeling scheme used in MS SQL Server 2005. It is similar to DeweyID, but uses only odd numbers for the first labeling [22]. When a new element is added, the element is labeled with an even number between

two odd numbers to concatenate another odd number. DLN(dynamic level numbering) [8] is another prefix-based labeling scheme integrated with eXist-db, another open source XML DBMS [2]. The size of prefix-based labels is usually larger than the one of interval-based labeling scheme. Thus, prefix-based labeling schemes have been also considered to be encoded in a concise form, for example, both ORDPATH and DLN represent their labels as compressed bitstrings.

In recent years, some efforts to parallelize XML query processing have been reported in literature. As multicore CPUs emerge, some researches have tried to utilize the multicore CPUs for parsing XML data in parallel [20, 23, 27]. However, they just focused on the parsing problem, not on XML labeling. HadoopXML is the first work which earnestly processes twig pattern queries for large scale XML data with the MapReduce framework [11]. This approach supports parallel and simultaneous processing of many queries for a very large size of XML document in a shared and balanced way. However, it also reveals that the current labeling schemes are all sequential so that labeling process delays its entire query processing time significantly.

MapReduce is a scalable and fault-tolerant data processing tool that is devised to process a large volume of data in parallel with many low-end computing nodes [13]. By virtue of its simplicity and fault-tolerance, MapReduce has been gaining significant momentum from both academia and industry. However, MapReduce has inherited limitations on its performance and I/O efficiency. Therefore, many studies have endeavored to overcome the limitations [18]. A MapReduce job may have a straggling task that delays the overall job execution. Thus, the MapReduce framework sometimes re-executes the straggling task on idle nodes speculatively [13]. However, the speculative reexecution does not alleviate the straggling problem if the problem is caused by data skewness. Rather, it exacerbates the phenomenon by running redundant tasks on multiple nodes. *LEEN* is devised to solve the straggler problem by providing fair key partitioning with the knowledge of key distribution [16]. *SkewTune* is an adaptive algorithm devised to mitigate the skewness problem [17]. It detects straggler tasks at runtime and then partitions their inputs, which are unprocessed so far into multiple pieces. The input pieces are then assigned to other idle tasks. However, both *LEEN* and *SkewTune* require to modify MapReduce internals. Moreover, *LEEN* is only effective when the number of keys is sufficiently large and the key distribution is not extremely skewed. *SkewTune* pays for an additional cost for coordinating workloads across tasks at runtime.

7 Conclusion

XML labeling is an essential operation for efficient XML query processing. In this article, we suggested parallel XML labeling algorithms for two prominent tree labeling schemes with MapReduce. The algorithms label a massive volume of XML data in parallel. However, XML documents may have the skewed distribution of element frequencies and quite a few number of distinct tag names. This makes some nodes lag with many XML elements to process. We also provided two optimization techniques for mitigating the data skewness problem. With the optimization techniques, our parallel labeling algorithms efficiently label a massive volume of XML data in a scalable and balanced way.

Acknowledgements This work was funded by the MSIP(Ministry of Science, ICT & Future Planning), Korea in the ICT R&D Program 2013. It is also partly supported by the NRF grant funded by the Korea government (No. 2011-0016282), and the IT R&D program of MKE/KEIT Korea [10041709, Development of Key Technologies for Big Data Analysis and Management based on Next Generation Memory].

References

1. BaseX: Processing and Visualizing XML with a native XML Database. Grun, C. and others. http://www.baseX.org/
2. eXist-db: Open Source native XML Database. Meier, W and others. http://exist-db.org/
3. Hadoop. Apache Software Foundation. http://hadoop.apache.org
4. Wikipedia (2013): Database download. Wikipedia Foundation, Inc. http://en.wikipedia.org/wiki/Wikipedia:Database_download
5. Bairoch A et al (2005) The universal protein resource (UniProt). Nucleic Acids Res 33(suppl 1), D154–D159
6. Baker BS (1985) A new proof for the first-fit decreasing bin-packing algorithm. J Algorithms 6(1):49–70
7. Beyer K, Cochrane R, Josifovski V, Kleewein J, Lapis G, Lohman G, Lyle B, Özcan F, Pirahesh H, Seemann N et al (2005) System RX: one part relational, one part XML. In: Proceedings of the 2005 ACM SIGMOD conference. ACM, New York, pp 347–358
8. Böhme T, Rahm E (2004) Supporting efficient streaming and insertion of XML data in RDBMS. In: Proc 3rd DIWeb workshop, pp 70–81
9. Bray T, Paoli J, Sperberg-McQueen C, Maler E, Yergeau F (1997) Extensible markup language (XML). World Wide Web J 2(4):27–66
10. Bruno N et al (2002) Holistic twig joins: optimal XML pattern matching. In: Proceedings of the ACM SIGMOD conference, pp 310–321
11. Choi H, Lee KH, Kim SH, Lee YJ, Moon B (2012) HadoopXML: a suite for parallel processing of massive XML data with multiple twig pattern queries. In: Proceedings of the 21th ACM CIKM conference, pp 2737–2739
12. Cormen TH et al (2001) Introduction to algorithms. MIT Press, Cambridge
13. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
14. Dietz P (1982) Maintaining order in a linked list. In: Proceedings of the 14th annual ACM symposium on theory of computing. ACM, New York, pp 122–127
15. Grün C, Gath S, Holupirek A, Scholl M (2009) XQuery full text implementation in BaseX. Database and XML technologies, pp 114–128
16. Ibrahim S, Jin H, Lu L, Wu S, He B, Qi L (2010) Leen: locality/fairness-aware key partitioning for mapreduce in the cloud. In: IEEE second international conference on cloud computing technology and science (CloudCom), 2010. IEEE Press, New York, pp 17–24
17. Kwon Y, Balazinska M, Howe B, Rolia J (2012) SkewTune: mitigating skew in MapReduce applications. In: Proceedings of the ACM SIGMOD conference. ACM, New York, pp 25–36
18. Lee K, Choi H, Lee Y, Chung YD, Moon B (2012) Parallel data processing with MapReduce: A survey. ACM SIGMOD Record 40(4):11–20
19. Li Q, Moon B (2001) Indexing and querying XML data for regular path expressions. In: Proceedings of the 27th VLDB conference, pp 361–370
20. Lu W, Chiu K, Pan Y (2006) A parallel approach to XML parsing. In: 7th IEEE/ACM international conference on grid computing. IEEE Press, New York, pp 223–230
21. Murthy R, Liu Z, Krishnaprasad M, Chandrasekar S, Tran A, Sedlar E, Florescu D, Kotsovolos S, Agarwal N, Arora V et al (2005) Towards an enterprise XML architecture. In: Proceedings of the ACM SIGMOD conference. ACM, New York, pp 953–957
22. O'Neil P, O'Neil E, Pal S, Cseri I, Schaller G, Westbury N (2004) ORDPATHs: insert-friendly XML node labels. In: Proceedings of the ACM SIGMOD conference. ACM, New York, pp 903–908
23. Pan Y, Lu W, Zhang Y, Chili K (2007) A static load-balancing scheme for parallel XML parsing on multicore CPUs. In: 7th IEEE international symposium on cluster computing and the grid, 2007. CCGRID 2007. IEEE Press, New York, pp 351–362
24. Rao A et al Hanborq Distribution with Hadoop. https://github.com/hanborq/hadoop

25. Schmidt A, Waas F, Kersten M, Carey M, Manolescu I, Busse R (2002) XMark: a benchmark for XML data management. In: Proceedings of the 28th VLDB conference. VLDB Endowment, pp 974–985
26. Scott ML, SCOTT ML (1998) Dewey decimal classification. Libraries Unlimited
27. Shah B, Rao P, Moon B, Rajagopalan M (2009) A data parallel algorithm for XML DOM parsing. Database and XML technologies pp 75–90
28. Suciu D (1992) Treebank: XML data repository
29. Tatarinov I, Viglas S, Beyer K, Shanmugasundaram J, Shekita E, Zhang C (2002) Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD conference. ACM, New York, pp 204–215
30. Zhang C et al (2001) On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD conference, pp 425–436