Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мережних технологій

Магістерська робота

Освітній ступінь: магістр

На тему: «MODELING DISTRIBUTED GENERALIZED SUFFIX TREES FOR QUICK DATA ACCESS»

Виконала: студентка 2 року навчання
Спеціальності
122 Комп'ютерні науки
Діденко Віра Олексіївна

Керівник Глибовець А.М.
док. техн. наук
Рецензент Олецький О.В.
доцент, к.н.

Магістерська робота захищена
з оцінкою_____
Секретар ЕК С.А. Мелещенко
«___» _____2022

Київ 2022

Ministry of Education and Science of Ukraine

NATIONAL UNIVERSITY OF “KYIV-MOHYLA ACADEMY”

Network Technologies Department of the Faculty of Informatics

APPROVED

Head of the Network Technologies Department

associate professor, doctor of mathematics

_____ G.I. Malaschonok

(signature)

“____” _____ 2021 year.

INDIVIDUAL TASK

For the Master thesis

For 2-year Master Degree student of the Faculty of Informatics

TOPIC: MODELING DISTRIBUTED GENERALIZED SUFFIX TREES FOR QUICK DATA ACCESS

Output data:

Text part content of Master thesis:

Individual task

Calendar plan

Abstract

Introduction

Section 1: Background On Suffix Trees And Their Application

Section 2: Suffix Tree Construction Algorithms

Section 3: Modeling A New Distributed Generalized Suffix Tree Construction Algorithm By Optimizing Era

Conclusion

References

Issue date “____” _____ 2021 year

Supervisor _____(signature)

Task received _____(signature)_

CALENDAR PLAN

Theme: MODELING DISTRIBUTED GENERALIZED SUFFIX TREES FOR QUICK DATA ACCESS

#	Stage Name	Deadline	Note
1	Acquiring Master thesis topic	01.09.2021	
2	Finding appropriate literature	20.09.2021	
3	Researching suffix tree construction approaches	30.09.2021	
4	Researching Generalized suffix trees	01.10.2021	
5	Analyzing existing suffix tree construction algorithms and their comparison	20.10.2021	
6	Researching the ERa suffix tree construction algorithm	01.11.2021	
7	Targeting weak areas of ERa and developing optimization approaches	01.12.2021	
8	Optimizing the data distribution technique	01.01.2022	
9	Optimizing the parallel subtree partitioning stage	01.03.2022	
10	Optimizing the parallel subtree construction stage	15.05.2022	
11	Calculating the new algorithm’s complexity	30.05.2022	
12	Developed algorithm implementation	10.06.2022	
13	Performance comparison tests and evaluation	14.06.2022	
14	Master thesis main section and conclusion	14.06.2022	
15	Master thesis analysis with the Supervisor	15.06.2022	
16	Master thesis improvement	20.06.2022	
17	Creation of the presentation	30.06.2022	
18	Master thesis analysis with the Supervisor	30.06.2022	
19	Defending the Master thesis	06.07.2022	

Student: Didenko V.O. “____” _____

Supervisor: Glybovets A.M. “____” _____

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

Generalized suffix trees [1,2] are applied in a vast majority of applications for accelerating string-based operations and processing text data more efficiently. With the increase in volumes of data that need to be processed, several approaches have been proposed to parallelize generalized suffix tree construction in order to speed up the building process. Still, when processing large alphabets where the input string exceeds the available memory capacity, constructing the corresponding generalized suffix tree becomes a challenge.

The aim of this work is to distribute generalized suffix tree construction, so the process is efficient in terms of time complexity and memory consumption. A distributed approach to constructing the suffix tree will allow working with large alphabets and very long strings that exceed the available memory capacity. In this work, an efficient and highly scalable algorithm for constructing generalized suffix trees on distributed parallel platforms was modeled. The experimental results proved that the modeled algorithm's efficiency is no less than the before known Elastic Range algorithm (ERa) [1] while out-performing ERa [1] on specific data.

INTRODUCTION

The suffix tree [1,2] is a fundamental data structure for processing string data and is used for numerous tasks, such as text processing and information retrieval. The suffix tree data structure stores a string by indexing all possible suffixes of that string and is applied in a variety of systems to provide swift operations on strings. For example, a widely used string operation is finding a substring in a string. A suffix tree can find the substring in $O(|substring|)$. In contrast, the same operation without it would take significantly longer, that is, $O(|string| + |substring|)$, considering that the searched string is in practice much longer than the requested substring. Therefore, suffix trees are utilized to accelerate string operations and work with strings more efficiently.

Considering the intense increase in volumes of data that need to be processed, constructing the corresponding suffix tree takes up a significant amount of time [3,4,5,6]. To solve the problem and optimize the suffix tree construction process, several parallel suffix tree construction algorithms have been proposed based on MPI [3]; however, they are limited in terms of scalability and fault tolerance when large data volumes are introduced. At the same time, the demand for effective algorithms for constructing suffix trees on distributed parallel platforms such as Hadoop [7] and Apache Spark [8] is constantly increasing.

This work introduces an efficient and highly scalable algorithm for constructing generalized suffix trees on distributed parallel platforms. The

modeled algorithm consists of two main stages: parallel subtree partitioning and parallel subtree construction.

During the first stage, a new data distributing strategy is implemented. Next, an efficient subtree division algorithm is proposed, which constructs the LCP-array [9,10,11,12,13] in a parallel manner, that is, parallel calculation of how many times a substring occurs. In order to improve load balancing and reduce data reading costs, an effective strategy for distributing tasks among the parallel processes is provided.

During the second stage, the parallel subtree construction uses the LCP-Range data structure and multilateral LCP-Merge sorting algorithm [9,10,11,12,13] for parallel construction of the LCP-array [9,10,11,12,13] Also, for comparison, a well-known algorithm for constructing suffix trees Elastic Range (ERa) [1] is taken into account and further modified to improve its capabilities to process strings with lengths that exceed the available memory and adapted to run on the Apache Spark platform [8]. The purpose of this work is to model an algorithm for constructing a distributed generalized suffix tree that is no worse in terms of efficiency than the previously known Elastic Range algorithm [1] and that outperforms it on specific data.

The rest of this work is organized in the following three sections: the first section *Section 1: Background On Suffix Trees And Their Application* provides background information on suffix trees and generalized suffix trees, their characteristics, and application; the next section *Section 2: Suffix Tree Construction Algorithms* provides a comparison of existing suffix tree construction algorithms along with their pros and cons as well as provides

an overview of the ERa [1] generalized suffix tree construction algorithm; finally, the third section *Section 3: Modeling A New Distributed Generalized Suffix Tree Construction Algorithm By Optimizing ERa* proposes an improved algorithm for constructing distributed generalized suffix trees and provides details on the applied techniques for outperforming the aforementioned ERa [1] algorithm - as a result, the modeled algorithm's complexity was calculated and the experimental performance comparison results were provided.

Section 1: Background On Suffix Trees And Their Application

1.1 Introducing Suffix Trees

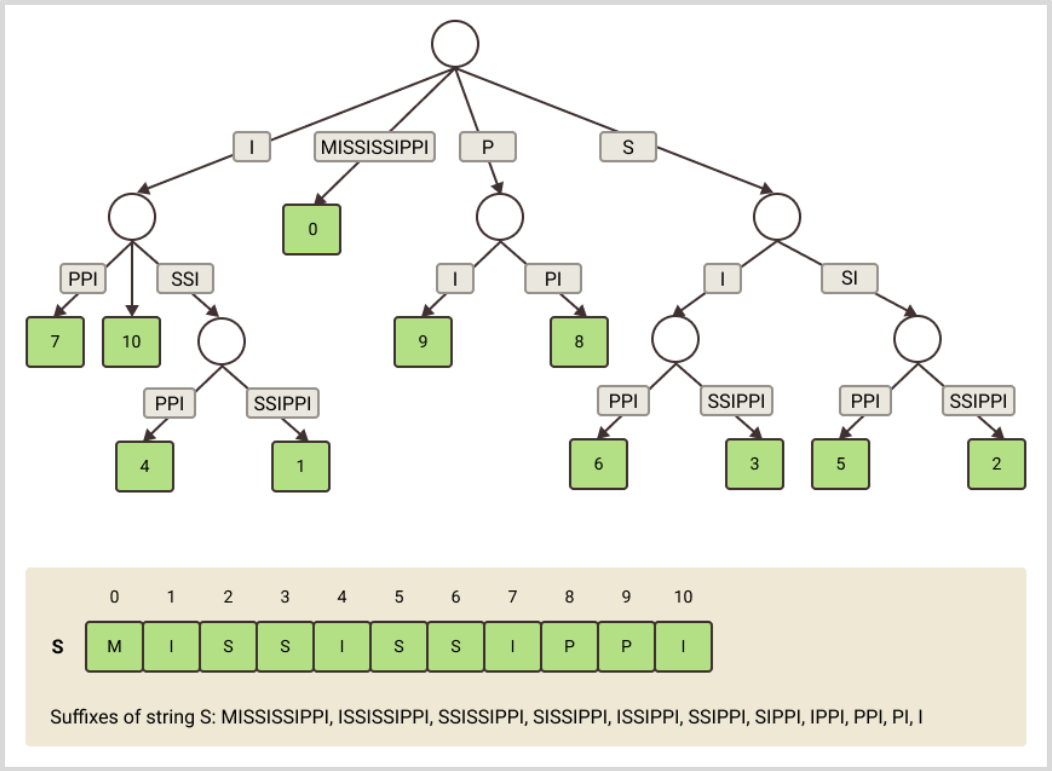

Figure 1: Example of the suffix tree data structure

The suffix tree [1], shown in *Figure 1: Example of the suffix tree data structure*, is a tree that indexes all suffixes in a string; the suffix tree is a fundamental data structure designed to swiftly process operations on strings, for example, phrase matching, finding the longest repeated substring in a character sequence, or revealing the maximum repeated phrase in a long sequence of characters [1,2]. A generalized suffix tree [1] is considered to be an extended version of a suffix tree, except this data structure stores not just one string but several - it indexes a set of suffixes of each string; therefore, the generalized suffix tree provides more flexibility for speeding up

various text-based operations. Being performance efficient when it comes to string processing, the generalized suffix tree data structure is widely incorporated in many applications to accelerate different text processing tasks, such as data compression, clustering of documents, etc.

For example, common text-based tasks that come across in the data analysis field, especially big data, are searching for a phrase or filtering, for instance, quickly performing the *begin* and *not begin* operation on a specific part of the text data. In such cases, if the complete data set (the string data collection) is indexed in a suffix tree, all lines in the data set that start with the searched phrase can be returned under $O(|searchedPhrase|$ time, which is much faster than if a suffix tree wasn't used which would have been $O(|entireStringDataCollection| + |searchedPhrase|)$ time. Similarly, a significant acceleration would be gained for other string operations such as phrase matching, finding the longest substring that is present in both strings, as well as finding all common substrings in a string data collection [1,2,14,15,16,17,18].

With the advent of big data, there is a growing demand for efficient algorithms for building a suffix tree for large alphabets [4,5]. However, building a suffix tree is a complex process that imposes the usage of intermediate data that takes up a significant portion of the available memory [6]. In addition, the process of building a suffix tree for massive data sets is a very time-consuming task [4,5,6].

To speed up the process of constructing the suffix tree and make it possible for data beyond the machine's available memory, a few parallel

algorithms were purposed that are designed to execute on several machines. Examples of such algorithms are WaveFront [1,4,5,18], PCF [1,16,18], and ERa [1], the latter is described in further detail in section *2.2 The Elastic Range (ERa) Algorithm For Suffix Tree Construction* of this work. However, it is important to consider that Both WaveFront [4,5,18] and ERa [1] duplicate the entire string data collection for each of the computing nodes, with each node processing the assigned subtree construction tasks [1,19], which obviously creates a scalability problem when massive data volumes are introduced.

Fast creation of suffix trees for an input string data collection is an extremely important task since, for instance, modern DNA sequencers can process multiple samples per hour, while financial programs generate continuous data streams that must be indexed. If the suffix tree can be stored in memory, there are already many quick and elegant solutions to build the suffix tree itself that will cope with the task in $O(|S|)$, such as the Ukkonen's algorithm [14], which maintains great performance, until the data goes beyond the memory available on the machine - in such case, its efficiency begins to drop. What is more, the built tree is in fact many times larger than the initial string data collection; the constructed suffix tree can increase the memory usage by more than 30 times [6].

At the same time, distributed parallel computing infrastructures such as Hadoop [7] and Spark [8] have become standards for large-scale data processing. They incorporate essential characteristics such as high scalability, ease of use, and sufficient fault tolerance. Unfortunately, most existing MPI-based parallel suffix tree algorithms [3] are unlikely to integrate

with the mentioned platforms. Attempts have been made to create a distributed suffix tree on these platforms, still, they impose the same limit in terms of scalability since the string data collection is duplicated for each computing node (as in WaveFront [1,4,5,18] and ERa [1]). Moreover, these approaches are not very effective for processing very long rows. Therefore, in this work, a new algorithm based on ERa [1] for constructing a generalized suffix tree on distributed parallel data platforms was created, which remains efficient while at the same time providing great scalability. The modeled algorithm supports indexing suffixes for a variety of long strings, ranging from one long string to several long strings of different lengths.

In this work, to perform distributed processing, all rows of the string data collection were merged into a single very long row - further referred to as the input string - and then the input string was partitioned into sections. Each partition (in other words, section) was assigned to a computing node and further processed. The rest of the algorithm consists of two main processing stages: parallel subtree partitioning and parallel subtree construction. The entire algorithm was modeled according to the following plan outline: 1. first, a new data-sharing strategy for subtree separation and subtree construction in the data parallelism paradigm was developed; 2. then, an efficient subtree distribution algorithm based on the parallel method of calculating the frequency of occurrence of the substring was introduced; 3. to improve load balance and reduce reading costs, an effective strategy for allocating subtree construction tasks was proposed; 4. at the subtree construction stage, a new LCP-Range [9] data structure and a versatile LCP-Merge [9] sorting algorithm for parallel LCP array [9] construction was provided; 5. the modeled prototype was tested on the Spark platform [5] and

the performance was experimentally evaluated and compared to the original ERa [1] algorithm implementation.

The next paragraph provides more background on generalized suffix trees and their characteristics.

1.2 Introducing Generalized Suffix Trees

For illustration purposes, consider the following string S equal to *BANANA* and consisting of the following suffixes: *A, NA, ANA, NANA, ANANA, BANANA*. To represent and save these suffixes in a suffix tree, we can construct a tree for string S, where $ denotes the terminal symbol. In addition, concatenating node values on the way from the root to a particular leaf results in one of the before-mentioned suffixes in string S; each leaf contains a value that indicates the starting position of the corresponding suffix in string S [1,2,3,4,5,6]. See *Figure 2: The suffix tree data structure for string S=BANANA* below.

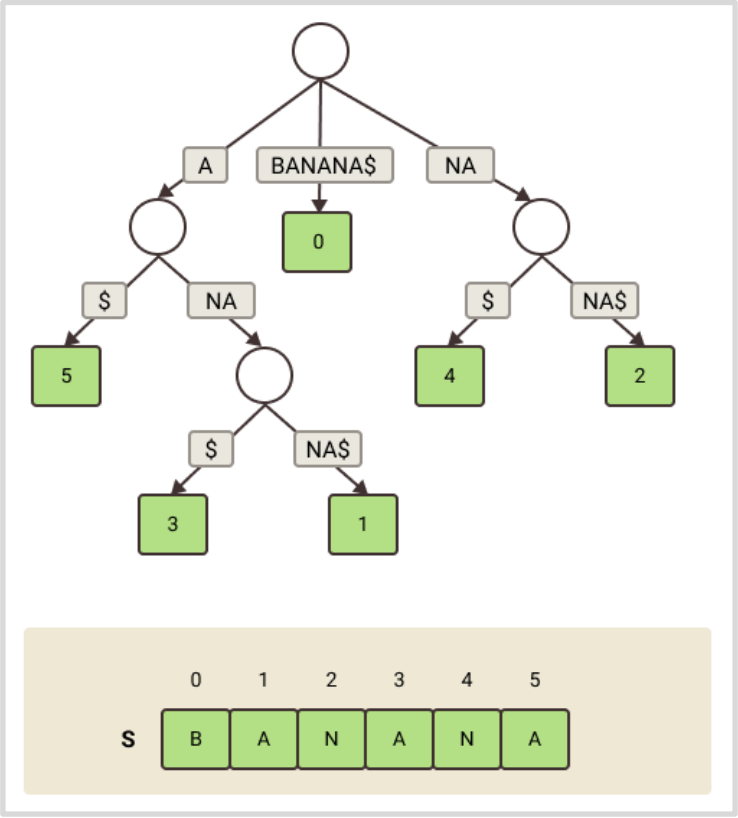

Figure 2: The suffix tree data structure for string S=BANANA

Next, for a general definition, let's denote a set of characters that will act as the alphabet: let $S = s0,s1...sn-1\ \$$, where $si \in \sum, i \in [0, n-1]$ and $\$ \notin \sum$. Then, the suffix S, which is denoted by Si, is a sequence of $si...sn-1\ \$, i \in [0, n-1]$; similarly, the prefix of the suffix Si is a sequence of $si...sj, j \in [i, n-1]$. In the rest of this work, the prefix of the suffix Si shall be referred to as the S-prefix. An important remark is that the terminal symbol $ is unique and guarantees that no suffix is a prefix of another suffix. Consequently, the suffix tree T is a tree that indexes all suffixes of string S. The properties and characteristics of a suffix tree are listed below as follows:

● Each internal node, except for the root node, has at least two child nodes, and each edge is denoted by a non-empty substring S.
● No two edges of a node can contain a value starting with the same initial symbol.
● There are exactly n leaves marked from 0 to n−1. For each leaf vi, with i ranging respectfully from 0 to n-1, concatenation of boundary values from the root to the leaf vi corresponds to the suffix Si [1,2,3,4,5,6].

Similar to a suffix tree, a generalized suffix tree is a tree that indexes all suffixes for a set of strings; the suffixes of each string are presented in the generalized suffix tree. A typical approach for constructing a generalized suffix tree for a set of strings is to add a unique terminal symbol (UTS) to each string and then merge all strings together and build a suffix tree for the merged string; in this case, the introduced terminal helper symbol that is supplemented in each string D is used to make sure that no suffix is a substring of another suffix. In this paper, all strings are merged directly without adding a terminal symbol to each string: instead, in this

implementation, information about the location of each string in the merged string, more specifically, the start and end position of each string in the merged string, can be used to distinguish between different strings in the merged string [1,2,3,4,5,6].

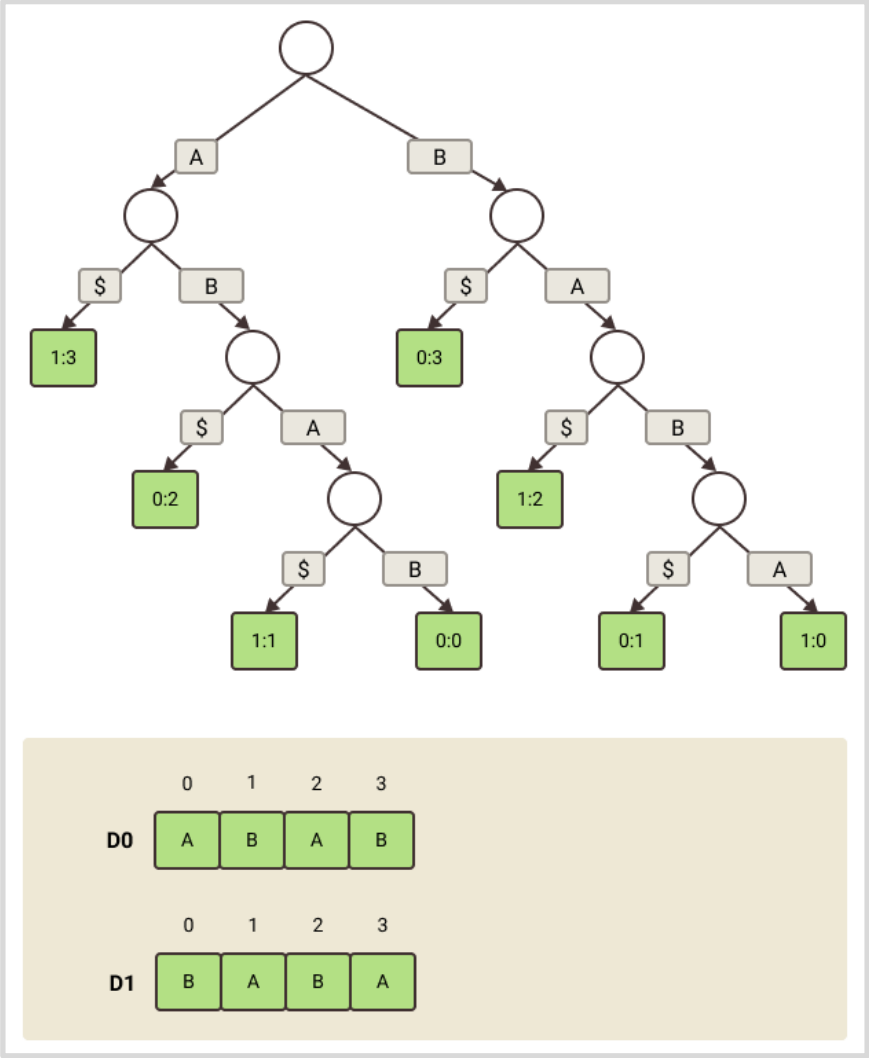

Figure 3: Example of the generalized suffix tree data structure

An example of a generalized suffix tree for a set of strings *D = {ABAB, BABA}* is provided above in *Figure 3: Example of the generalized suffix tree data structure*. Note that the leaves of the tree contain a value consisting of

two parts: the string ID and the suffix starting position. Considering that a suffix tree is a special case of the generalized suffix tree where the size of the set of strings D is equal to one (meaning a single string is being stored), it can be concluded that the algorithm for constructing a suffix tree also applies to the generalized version making it possible to focus on constructing a suffix tree for one long string [1,2,3,4,5,6].

Section 2: Suffix Tree Construction Algorithms

2.1 Existing Algorithms Analysis And Comparison

Suffix tree construction algorithms are divided into three main categories: in-memory (those that store data entirely in RAM), disk-based (those that store data on one machine), and, finally, those that are distributed. The following paragraphs go over each category, analyzing and comparing the corresponding algorithms against the following criteria: complexity, memory usage, string access, and the ability to be parallelized.

Category	In-memory	Disk-based	Distributed
Algorithm	Ukkonen	TRELLIS	WaveFront
Complexity	O(n)	O(n^2)	O(n^2)
Memory usage	Poor	Good	Good
String access	Random	Random	Sequential
Can be parallelized	No	No	Yes

Table 1: Existing suffix tree construction algorithms comparison

Approaches that store everything in memory perform well and fast, for example, the Ukkonen suffix tree construction algorithm [14]; as shown in *Table 1: Existing suffix tree construction algorithms comparison* above, they have linear complexity as long as the input string can be stored in memory. However, as soon as it is required to save an intermediate result on the disk, the performance drops and such algorithms become inefficient.

The problem of limited memory is solved by algorithms that allow you to dump intermediate results to the disk, by dividing the tree into subtrees that are stored separately - the disk-based suffix tree construction algorithms, such as TRELLIS [15]. If necessary, the required part can later be retrieved by easily lifting it in memory. When the correct suffix is found, the corresponding subtrees are combined into the appropriate full suffix tree. The process of building subtrees can be easily parallelized, but the final step that requires merging the subtrees into a complete tree requires good coordination and is already quite difficult to parallel.

The last category includes algorithms that work on different machines, in other words, that are distributed. For instance, an example of a well-known distributed algorithm is WaveFront [1,4,5,18]. WaveFront [1,4,5,18] works with the entire row on independent parts of the resulting suffix tree. The suffix tree is partitioned using variable-length S-prefixes, making sure that each subtree is currently in memory. Because the string S may not be in memory, the algorithm may need to read the input string S several times. To minimize the cost of the read operation, WaveFront [1,4,5,18] addresses the string S strictly in sequential order. In addition, each subtree is processed independently without the merge phase, so the algorithm is easily parallelized [1,4,5,18].

2.2 The Elastic Range (ERa) Algorithm For Suffix Tree Construction

Let us consider the ERa algorithm [1] that is noted for being able to process very long strings and work with large alphabets; aside from performing just as fast as the existing algorithms that are designed for a limited amount of memory, ERa [1] is also very easy to parallelize.

The algorithm considers the machine's memory capacity and divides the task of constructing the suffix tree vertically by partitioning the given suffix tree into independent subtrees, ensuring that the given tree can be stored in memory and does not exceed the available memory [1].

The resulting subtrees are then divided horizontally into parts, in such a way that the resulting parts can further be processed in memory in one operation. At the same time, at each step, the horizontal division changes and adjusts to the shape of the tree to maximize memory usage. The ERa algorithm [1] goes even further and combines the subtrees in order to divide the read operation cost when the input string S is read from the disk storage [1].

In addition, horizontal partitioning is applied independently to each subtree from top to bottom; the ERa algorithm [1] dynamically adjusts the width of each horizontal partition based on how many paths in the subtree are still undergoing processing which enables ERa [1] to allocate only a small amount of memory for each buffer, making the algorithm cache-friendly and at the same time minimizing overhead cost. Moreover, each group

represents an independent unit - groups can be processed sequentially or in parallel. The resulting subtrees are then collected to represent the final suffix tree that is extremely lightweight and consists of S-prefixes, which are used for vertical partitioning [1].

2.2.1 Vertical Partitioning

To describe the vertical partitioning step, the following terms are introduced. Let p be the S-prefix, Tp - the subtree corresponding to the S-prefix p, and Fp - the number of suffixes corresponding to the S-prefix p. Considering that each suffix corresponds to a node leaf in the subtree Tp, it can be derived that the number of internal nodes corresponds to the number of leaves which takes up the following number of bytes in size: $Tp - 2 \times Fp \times sizeOf(treeNode)$ bytes [1].

Next, let M denote the amount of memory reserved for the subtree, meaning that the subtree Tp can fit in memory only when the number of suffixes Fp (that correspond to the S-prefix p) is less than or equal to FM: $Fp <= FM$, with FM being calculated as: $\frac{M}{2 \times sizeOf(treeNode)}$, where M denotes the amount of reserved memory for the subtree [1].

In order to split the tree T into subtrees that can be placed in the reserved memory M, the idea of S-prefixes of variable length is utilized. The algorithm begins by creating a workgroup that contains only one S-prefix for each character of the alphabet. Then the entire input string S is scanned to calculate the frequency of occurrences of each S-prefix for each character of the alphabet. At the end of this step, each prefix that occurs with a frequency that is at least equal to FM is removed from the workgroup. Each remaining prefix is increased by one, and the process is then repeated until the workgroup becomes empty [1].

However, this approach has a drawback since the resulting trees become unbalanced. When parallelized, we are faced with a significant disadvantage: certain nodes that contain small subtrees could remain idle at the same time while other nodes are overloaded calculating large parts of subtrees obtained after partitioning. In order to solve this problem, at the end small trees should be grouped into a single group - this will optimize the work of nodes during parallelization and further reduce the cost of disk read operations that are known to be computationally intensive. The resulting algorithm for vertical partitioning is provided in the code snippet below [1]:

```
Input: string S, alphabet A, frequency threshold FM
Output: tree set VirtualTrees
VirtualTrees := nul
P := nul // final S-prefix list
P’ := (for each symbol s ∈ A do generate S-prefix pi = s)
repeat
   scan the input string S
   fp := calculate occurrence frequency of each S-prefix pi in P`
   for each pi in P` do
      if ( 0 < fpi<= FM ) then add pi to P
     else (for each character s ∈ A do add pis into P’ )
      remove pi from P’
until P` = nul
sort P in reverse fpi order
repeat
    G := nul //S-prefix group in tree
    add P.head into G and remove it from P
    next := next element in P
    while not end of P
      if fits into the available threshold for construction then
         add next into G and remove the corresponding element from P
     next := next element in P
```

```
23      add G into VirtualTrees
24  until P = nul
25  return VirtualTrees
26
```

2.2.2 Horizontal Partitioning

During the horizontal partitioning step, the ERa algorithm [1] constructs a suffix subtree Tp for the S-prefix p, where Tp fits into the available memory. This process utilizes properties of the suffix tree that were not used in previous approaches: first of all, access to the tree on the disk is optimized; next, the tree is modified to save even more memory [1].

The preparation stage is performed using the *SubTreePrepare* algorithm [1], which extends the optimized version of *BranchEdge* [1] that is required to scan the input string and construct a subtree and uses a new intermediate data structure instead of using a subtree for the subtree construction process [1].

The construction stage is performed using the *BuildSubTree* algorithm [1], which uses the data structure created during the preparation phase to create the suffix subtree. By separating the subtree construction stage from the preparation stage, we localize memory access and, therefore, avoid costly traversals of the partial subtree for each new node [1].

The essence of the proposed method is the intermediate data structure, consisting of the following two arrays: array L for storing the tree leaves, and array B for storing information about the tree branches. More precisely, the array L retains the position of the input S-prefix p in S, that is, the leaves of the subtree [1].

Next, we will focus on developing an algorithm for constructing outside the main suffix tree on distributed parallel data platforms, which are in turn standards for big data processing due to their high scalability, ease of use, and sufficient fault tolerance.

Section 3: Modeling A New Distributed Generalized Suffix Tree Construction Algorithm By Optimizing ERa

Existing two-step suffix tree construction algorithms, such as WaveFront [1,4,5,18] and ERa [1], do not provide sufficient scalability when very long strings are introduced: first, at the subtree partition stage, the frequency count for S-prefixes must be calculated from the input string several times in succession; then, during the subtree construction phase, both WaveFront [1,4,5,18] and ERa [1] duplicate the input string in each computing node [1,4,5,18]. Therefore, it can be foretold that as the size of the input string increases, these algorithms will become ineffective. To solve the problem of scalability, this work introduces a new algorithm based on ERa [1] by optimizing the approaches used in ERa [1] to achieve better performance. The optimized new version of the generalized suffix tree construction algorithm proposes efficient distributed parallel algorithms for both the subtree partitioning and subtree construction stages. This section describes the new algorithm's implementation in further detail and consists of the following steps: first, each stage of the suffix tree construction algorithm is explained in detail which includes three main parts - data distribution, parallel subtree partitioning, and parallel subtree construction; next, the modeled algorithm's complexity is calculated; finally, the experimental performance test results are provided.

3.1 New Algorithm Preparation Stage - Data Distribution

To partition and construct the subtree in the data parallel paradigm, first, a new data distribution strategy was developed and is described in detail in this section.

To start, in order to ensure balance among the computing nodes, the input string is divided into even parts, and each partition is assigned to a computing node. Because the partitions are processed in parallel, an additional tail is assigned to each partition, that is, to each divided part (except for the last partition - there is no tail there) to ensure that the parallel S-prefix frequency counting for splitting the subtree as well as the parallel matching of the S-prefix for constructing the subtree are both done correctly [19].

For a better understanding, let us assume that the input string is divided into k sections, with each section being equal. For $i \in [0, k-1)$ the split tail $|i|$ is the range of initial split symbols $|i+1|$. The additional tail that is added to each partition serves the following purposes: firstly, it ensures the correctness of the parallel frequency calculation for each S-prefix; secondly, it helps guarantee that the S-prefix is correct during the parallel subtree construction stage. The next two paragraphs elaborate more on both of the mentioned points.

For instance, to illustrate how the tail ensures the correctness of the parallel frequency calculation for each S-prefix, the following example is provided. When partitioning the subtree, the S-prefix frequencies of a certain

length need to be calculated. In this example, it is assumed that the length of the counting window w is equal to four. Suppose the additional tail *(CGG)* of the first partition is not taken into account. In that case, S prefixes with a length of four in the initial positions from eight to ten will not be found - this will lead to incorrectly calculated frequency counts of *GTGC*, *TGCG*, and *GCGG*. For this reason, the first partition must contain an additional tail, the length of which is at least equal to three. Hence, the tail attached to each input partition plays an important role in ensuring the correctness of the parallel frequency calculation [19].

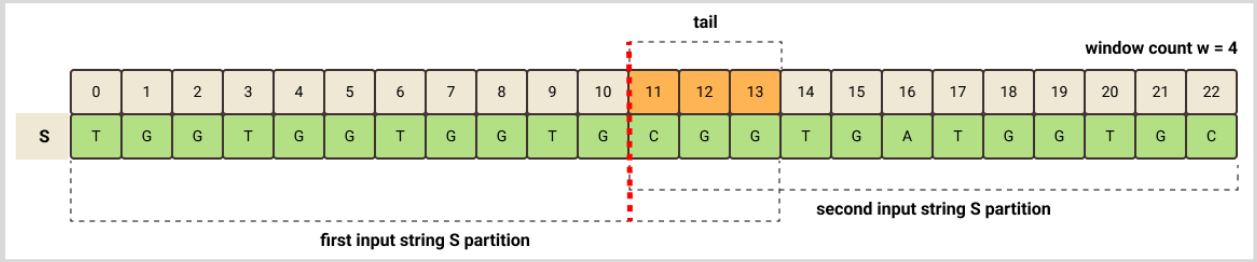

Figure 4: The additional tail that is assigned to each partition

Next, it is shown how the additional tail is also used to guarantee the correctness of the parallel S-prefix when the subtree is being constructed. When the subtree is being constructed, it is important to find all occurrences of the S-prefix; note that the coincidence of the S-prefix occurring in each partition is also processed in parallel. Assuming that the input S-prefix is *TGC*, as shown in *Figure 4: The additional tail that is assigned to each partition* above, without an additional tail, the S-prefix TGC at the starting position 9 will not be found in the first cleavage. Thus, the tail attached to each input partition plays an important role in ensuring the correctness of the parallel matching of the S-prefix.

Still, the key problem when distributing data is correctly determining how long the tail for each partition should be. The frequency threshold, marked as FM, is determined using a similar approach as used in the ERa algorithm [1]. This frequency threshold should not be exceeded by an S-prefix frequency from the set [1]. Next, to guarantee that the parallel frequency is calculated correctly and that the S-prefix is matched in the parallel paradigm without error, it is evident that the tail length must be at least the same length as the maximum length of the S-prefixes in the set which will further be referred to as *maxPrefixLen*. On average, the *maxPrefixLen* value can be calculated based on the fact that the initial string is derived from a random symmetric Bernoulli distribution [1,17]. In this scenario, the frequency threshold is equal to the following: $\frac{totalSizeOfInputString}{alphabetSize} \times maxPrefixLen.$

Using this equation, the *maxPrefixLen* can be represented by the following formula: $\log_{alphabetSize} \frac{totalSizeOfInputString}{FM}$

An important remark regarding the tail length is that each partition is assigned an additional tail of the same length when constructing a suffix tree; however, when a generalized suffix tree is created, the length of the additional tail could differ across partitions. Such a difference occurs since, by design, the generalized suffix tree can store more than one string - each partition created during the data partitioning step can potentially have several input strings. Taking the latter into account, the tail length for the input partition can be calculated in the following way: $\min(len(RC), TAIL\_LEN)$,

where RC is the content of the remaining last string in the input partition in the subsequent partitions and TAIL_LEN is the default tail length.

3.2 New Algorithm - Parallel Subtree Partitioning Stage

After the data preparation step that was described in the previous section, where the input string was split into parts otherwise referred to as partitions, the distributed suffix tree was created. The process of creating the final suffix tree consists of the following two main stages: the subtree partitioning stage and the subtree construction stage. This section goes over the key aspects of the subtree partitioning stage.

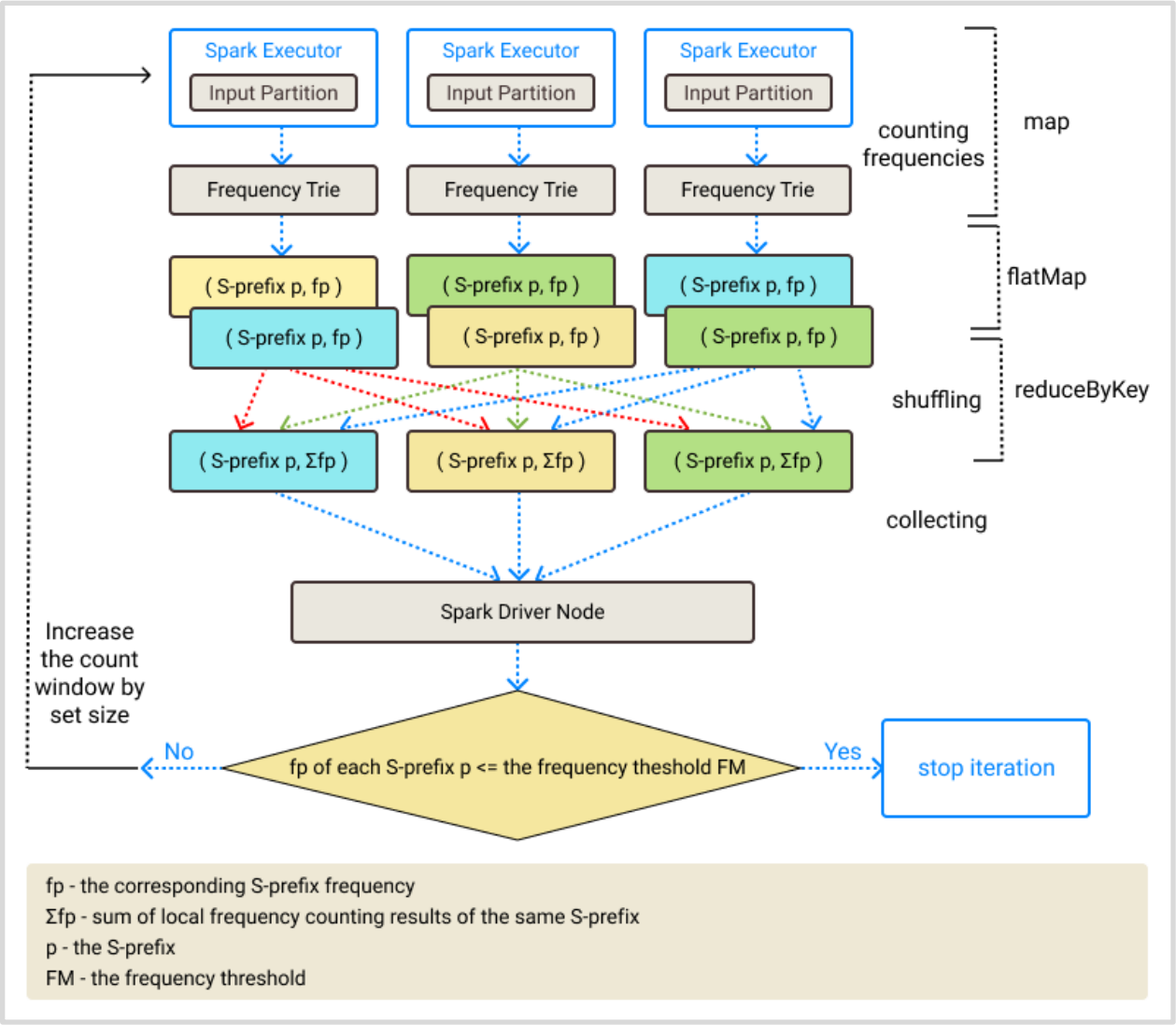

Figure 5: Overview of the parallel subtree partitioning stage algorithm

The subtree partitioning stage is based on parallel frequency counting: at the start, each computing node independently accesses the partition and constructs a local frequency trie; next, the corresponding S-prefix p together with its frequency fp is obtained for each leaf node of the local frequency trie, and the next step shuffles all the gathered pairs of the S-prefix and the corresponding frequency (p, fp): the pairs (p, fp) are shuffled so that the same computing node receives the results of calculating the local frequency of the same S-prefix; when the local frequency calculations of the same S-prefix are sent to a computing node, they are independently aggregated and passed to the driver node which gathers the results and decides if the subtree partitioning process is complete and can therefore be stopped. The described process running on Spark [8] is illustrated in *Figure 5: Overview of the parallel subtree partitioning stage algorithm* above. The next paragraphs provide further details on the following key techniques that were used for partitioning the subtree in a parallel manner: parallel frequency counting, the frequency trie [9], and the incremental count window approach [19].

Calculating the occurrence frequency for the S-prefix is the main step in splitting the subtree: considering that the input string is split into partitions (during the data partitioning stage), the frequencies can be calculated in each partition in parallel. Moreover, the before-mentioned additional tail that is assigned to each partition provides a guarantee that the parallel frequency calculation has been performed correctly.

In addition, to efficiently calculate the frequencies of S-prefixes, a frequency trie data structure [9] was implemented. The frequency trie is a tree-like data structure that stores the S-prefix frequency calculated from a

given input string S and is further defined as follows: first, the edge of the frequency trie and the node of the frequency tree shall be marked as e and v respectfully; label(e) indicates the character that occurred in S; path(v) indicates the concatenation of edge values on the path from the root to the node v; and label(v) indicates the frequency with which path(v) occurs in string S. Therefore, it can be seen that the S-prefix p in the frequency trie is represented by path(v), and fp is actually the label(v) in the frequency trie. By definition, label(v) can be viewed as the summed-up labels of all child nodes if node v is an internal node; in the same way, using the path information and the label of all leaf nodes, the frequency trie can be reconstructed. The frequency trie data structure is illustrated in *Figure 6: The frequency trie data structure* below:

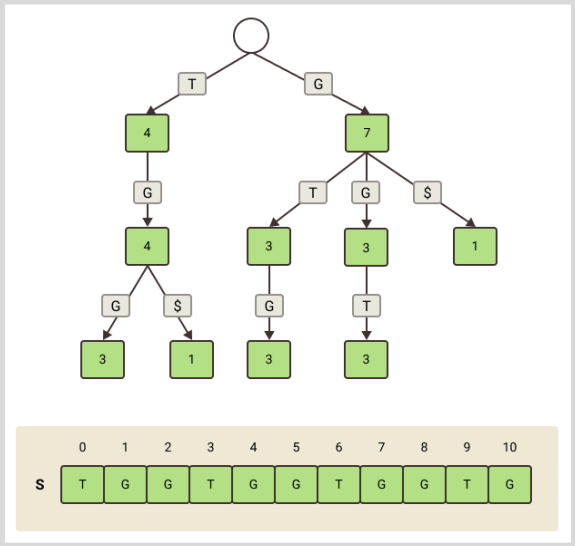

Figure 6: The frequency trie data structure

Considering this frequency trie characteristic, it is possible to reduce the total number of iterations (that is, Input/Output accesses) by calculating the

frequencies of different lengthed S-prefixes by simply scanning the input string.

The last applied technique for partitioning the subtree is the incremental window count illustrated in *Figure 5: Overview of the parallel subtree partitioning stage algorithm*. The general process of subtree partitioning in a parallel manner is organized in multiple iterations, during which the count window increases by the defined step size. Let winit denote the initial count window; consequently, during the first iteration, the frequencies of all winit lengthed S-prefixes are counted. When the first iteration completes, the S-prefixes are added to either set P or set R depending on whether the frequency of the S-prefix fits in the FM threshold or not. The driver node checks whether the R set contains S-prefixes with frequencies greater than the FM threshold - if so, the R set is broadcasted to each performer to be processed further; if R is empty, the whole process completes, and the iteration is terminated. In case of the next iteration taking place, the window size is increased by the defined step size, denoted by wstepSize, and, in this case, let the increased count window be denoted by wnext; consequently, the frequencies of all S-prefixes that have a length that fits into the [winit, wnext] interval are calculated in this iteration. As mentioned before, the iteration process stops when the R set does not contain any more S-prefixes with frequencies exceeding the FM threshold [19].

At the final step, a frequency trie is constructed using the S-prefixes in the resulting set P, and redundant nodes are removed by trimming all child nodes of a node in the frequency trie that has a label that does not exceed

the FM threshold [19]. The code snippet below provides the implementation of the tree node obtained after subtree partitioning:

```
internal class TreeNode : ISerializable
{
    var parent;
    var edges = new HashMap();
    var info;
    var count = 0;
    var shouldStop = false;

    public TreeNode() : this(null, null) { }
    public TreeNode(Info info) : this(null, info) { }
    public TreeNode(TreeNode node) : this(node, null) { }
    public TreeNode(TreeNode node, Info info)
    {
       parent = node;
       this.info = info;
    }
    public AddIndex(Pair pair){
     if (info == null){
       info = new Info();
     }
     info.suffixIndexes.AddIndex(pair)
    }
    public AddTerminalIndex(Pair pair){
       if (info == null){
          info = new Info();
       }
       info.terminalSuffixIndices.AddIndex(pair)
    }
    public AddTerminalIndex(List list){
       if (info == null){
          info = new Info();
       }
```

```
33          info.terminalSuffixIndices.AddIndex(list)
34      }
35  }
```

3.3 New Algorithm - Parallel Subtree Construction Stage

After the subtree partitioning stage that was described in the previous section, the subtree construction stage took place that consisted of the following steps: task allocation for subtree construction, local suffix sorting, LCP-Merge [9] sorting, and subtree construction. An overview of the parallel subtree construction process is depicted below in *Figure 7: Overview of the parallel subtree construction stage algorithm*:

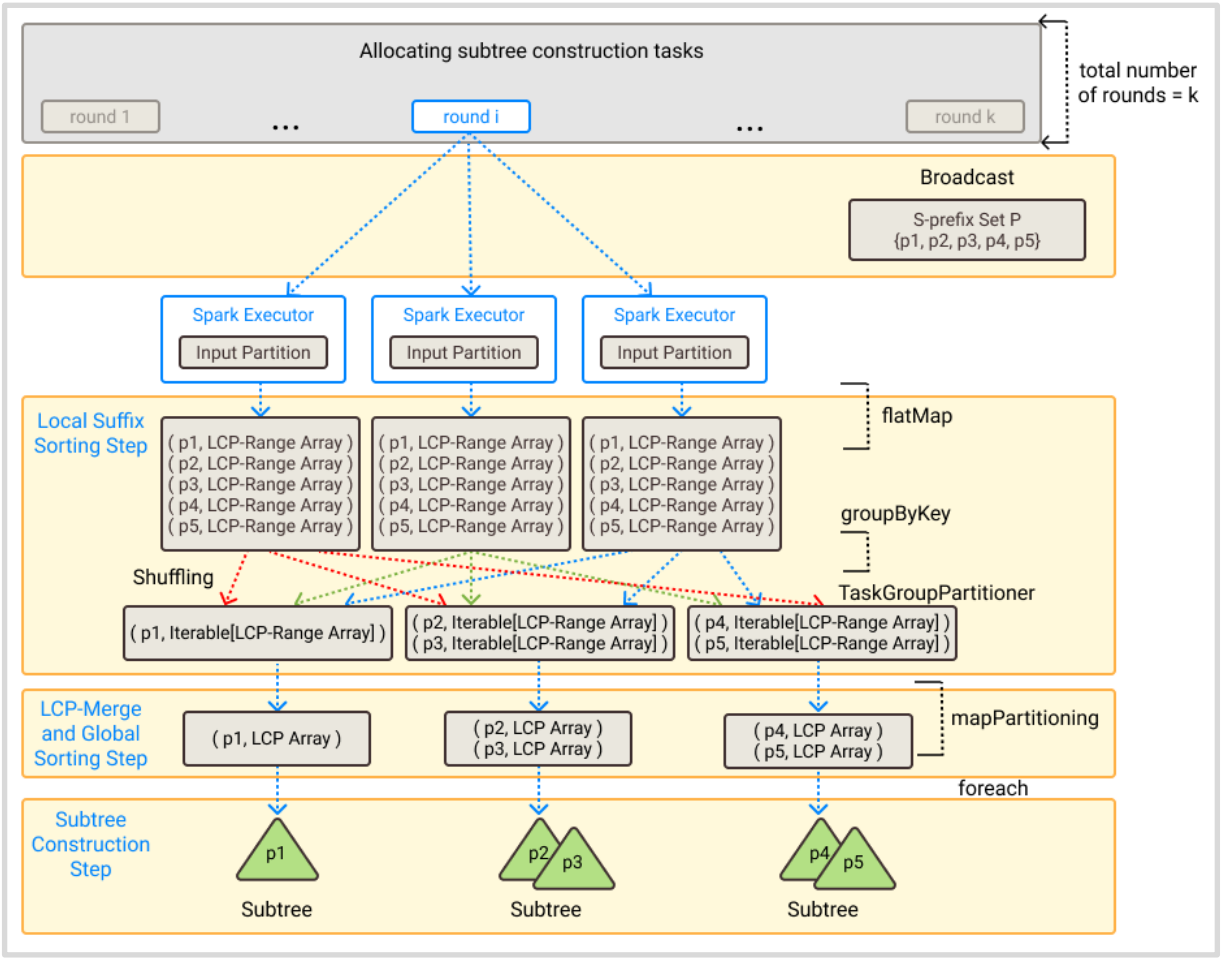

Figure 7: Overview of the parallel subtree construction stage algorithm

At the start, for constructing the subtree in parallel, subtree construction tasks were distributed among computing nodes; afterward, each subtree was

constructed using a new multifaceted algorithm based on LCP-Merge [9]. The following paragraphs in this section go over the key aspects of the subtree construction steps in further detail. For a better understanding, the key algorithms - the LCP-Range aware comparison algorithm [9] and the LCP-Merge Sorting [9] implementation are provided below:

The LCP-Range aware comparison algorithm [9,19]:

```
Input : String s1 and s2 LCP-Range of s1 to reference sr : lr1,
LCP-Range of s2 to reference sr : lr2.
Output : ( sm, lg, lr ), lr is the LCP-Range of lg (larger) to sm
(smaller)
//Compare offsets:
if lr1.offset > lr2.offset then
    return ( s1, s2, lr2 )
if lr1.offset < lr2.offset then
    return ( s2, s1, lr1 )
//Compare ranges:
range1Len = len( lr1.range )
range2Len = len( lr2.range )
rangeMin = min( range1Len, range2Len )
for i = 0 to rangeMin do
    if lr1.range[i] > lr2.range[i] then
        lrnew = ( lr2.range[i], lr1.range[i, range1Len − 1], lr2.offset + i )
        return ( s2, s1, lrnew )
    if lr1.range[i] < lr2.range[i] then
            lrnew = ( lr1.range[i], lr2.range[i,range2Len − 1], lr1.offset + i )
        return ( s1, s2, lrnew )
// s1 and s2 cannot be compared based on LCP-Range:
updatedOffset = − ( rangeMin + lr1.offset )
lrunkown = ( 0, {0}, updatedOffset )
return ( s1, s2, lrunkown )
```

```
26
```

The LCP-Merge Sorting algorithm overview [9,19]:

```
Input : S-prefix p, Set of LCP-Range arrays L
Output : Partially ordered LCP array PO_LCP
// Partially ordered LCP array PO_LCP
PO_LCP = {}
// k-way LCP-Range array
k = length( L )
// The loser tree
loserTree = createLoserTree( k )

firstPlayer s = getAllFirstPlayers( L )
smallest = init( L, firstPlayers )
add ( smallest.loc, ( 0 , 0 , 0 )) to PO_LCP
nextWayIndex = smallest.wIndex
repeat
   player = getNextPlayer( L[nextWayIndex] )
   if player is NULL then
      //This way has no elements
        k = k - 1
      player = makeEndPlayer( nextWayIndex )
      smallest = rebuild( loserTree, player )
   if smallest.loc is not NULL then
      smallest.lcp = transform( smallest.lcpRange )
      add ( smallest.loc, smallest.lcp ) to PO_LCP
      nextWayIndex = smallest.wIndex
until k = 0;
return PO_LCP
```

The following paragraphs explain the algorithms in further detail with examples.

First, it is important to point out the main factors that impact the task allocation step's performance and how these factors were addressed in this work to improve the overall performance of the resulting parallel subtree construction stage. Mostly, the distribution of subtree construction tasks is affected by two main factors: the cost of input/output calls and keeping the worker nodes balanced. The first factor can be explained by the fact that each task must access the entire input string when the subtree is being built - if there are too many small subtrees to process, the reading costs will be high. As for the second factor, the subtrees can differ in size - if certain computational nodes get overloaded while other nodes remain idle, it would affect the overall performance. Therefore, a strategy for distributing tasks with a balanced workload should be developed.

To solve both of the mentioned above problems, this work proposes a fairly new task distribution strategy for constructing the subtree, which implies using two algorithms in combination: the Bin-Packing algorithm [20] and the Number-Partitioning algorithm [21,22].

The first problem - reducing the cost of reading a subtree structure - can be addressed by grouping as many subtrees as possible, similar to ERa [1], which will reduce the total number of read operations. Considering that each formed group of subtrees has a total sum of S-prefixes that is not greater than the FM frequency threshold, all subtrees in a group can be constructed in one computing task. Minimizing the number of subtree groups can be viewed as a container packaging task, this is where the Bin-Packing algorithm [20] can be applied, where Bin is an S-prefix group and the

capacity of Bin corresponds to the FM threshold - the Bin-Packing algorithm [20] solves the problem of finding the minimum number of groups that can be formed taking into account that the sum of the frequencies in each group is not greater than the FM threshold.

At the same time, the Bin-Packing algorithm [20] cannot guarantee that for each group the load will be balanced. The problem of load balance can be considered as a number distribution task [21,22]. The number corresponds to the frequency of the S-prefix. Thus, by applying the Number-Partitioning algorithm [21,22]. (algorithm for dividing numbers) it can be ensured that the sum of frequencies in each group is as equal as possible. Still, the algorithm for dividing numbers must know the total number of groups in advance [21,22]. Hence, the Bin-Packing algorithm [20] and the Number-Partitioning algorithm [21,22]. were used in combination in the parallel subtree construction stage's final implementation.

Furthermore, before proceeding to the parallel subtree construction implementation details, it is important to define the total number of subtree construction rounds. In practice, the degree of parallelism p can be considered as the number of performers on the Spark platform [8,19]. Considering that the input string had been divided into k partitions, the number of groups of subtree construction tasks is equal to k multiplied by p. Thus, it can be derived that the distribution of all tasks to build a subtree consists of k rounds.

As calculated in the previous paragraph, the parallel subtree construction stage takes k rounds in total to complete. In each round, all

Spark Executors [8] process p groups of subtree construction tasks in parallel. At the start of each round, all involved S-prefixes are broadcasted to each Spark Executor [8] - worker nodes' processes. Next, in order to generate a local LCP-Range array [9] for each S-prefix, the suffixes are locally sorted on each executor in the following manner: each Spark Executor [8] is given an input partition (a part of the split input string); using the input partition, each executor starts by matching all S-prefixes in the corresponding input partition; for instance, for an S-prefix p, all suffixes that start with p in each input partition are sorted. Based on the sorted suffixes, the local LCP-Range array [9,19] is then constructed. The LCP-Range array [9] construction implementation is provided in the code snippet below:

```
public LCPRangeElement[] GenerateLCPRange(SegmentprefInfo prefInfo)
{
    var suffArray = new SharedBufferSuffixArray(
        prefInfo.Loc.ToArray,
        true
    );
    var res = new LCPRangeElement[](suffArray.Number);
    res[0] = new LCPRangeElement(
        new Location(suffArray[0].Location, prefInfo.Length),
        0,
        suffArray[0].GetSegment(_bitsPerItem, prefInfo.Length)
    );
    res[0].Content = suffArray[0].ToArrayAndTake(
        prefInfo.Length,
        _firstBuf
    );
    res[0].Lcp = (0, new byte[], 0);
    res[0].ExistRemainItem = if(suffArray[0].Length -
prefInfo.Length <= Math.Max(_segmentLength, _firstBuf)) false else
true;
```

```
21      var i = 1;
22      var lcpMax = 0;
23      while (i < res.Size) {
24         var segment = suffArray[i].GetSegment(
25             _bitsPerItem,
26             prefInfo.Length
27         );
28         var lcp = suffArray.LcpForBuild(
29             i,
30             _range,
31             prefInfo.Length
32         );
33         lcpMax = Math.Max(lcpMax, lcp.Location3);
34         var loc = new Location(
35             suffArray[i].Location,
36             prefInfo.Length
37         );
38         var remain = false;
39         if (lcp == null) remain = false;
40         else {
41            var LcpExtraLength 0 ;
42            if (lcp.Location2.Length != 0) LcpExtraLength =
43 lcp.Location2.Length - 1
44            if (LcpExtraLength + lcp.Location3 >= suffArray[i].Length
45 - prefInfo.Length) remain = false
46            else if (suffArray[i].Length - prefInfo.Length <=
47 _segmentLength) remain = false
48            else remain = true
49       }
50       res[i] = MakeLCPRangeElement(loc, 0, segment, lcp, remain);
51       i += 1;
52    }
53    return res;}
```

After the local suffix sorting step and LCP-Range array [9] construction, the LCP-Range arrays [9] of the same S-prefix are grouped together. At the same time, S-prefixes that are part of the same subtree construction task group are located in the same partition, hence, all subtrees in the same group can be built during a single computing task. During the LCP-Merge [9] sorting step, all local LCP-Range arrays [9] of the same S-prefix are combined by utilizing the LCP-Range aware comparison method [9] - as a result, a globally ordered LCP array [9] is created. The next paragraph provides further details on the LCP-Merge [9] sorting step and the LCP array [9] generation since the aforementioned LCP-Range aware comparison method [9] is essential for constructing the LCP array [9,10,11,12,13].

The parallel LCP-Range array [9] used for the final subtree creation was constructed using the LCP-Range [9] data structure and the LCP-Range aware comparison method [9]; the overall construction process consisted of the following main stages: local suffix sorting, LCP-Merge [9,10,11,12,13] sorting, and sorting at disordered intervals. An LCP-Range [9] data structure was created for each suffix in the sorted suffix array.

The LCP-Range data structure is illustrated below in *Figure 8: The LCP-Range data structure*. As shown, the LCP-Range data structure stores information for comparing two strings and is represented by an element consisting of three parts: the first character of the smaller string where both strings start to differ, a sequence of characters of the larger string ranging from the index where both strings start to differ until the specified range length, and the offset number - the index where both strings start to differ -

hence, the defined triplet will be denoted in the following way: *lcp_range(s1, s2) = (c, range, offset)*.

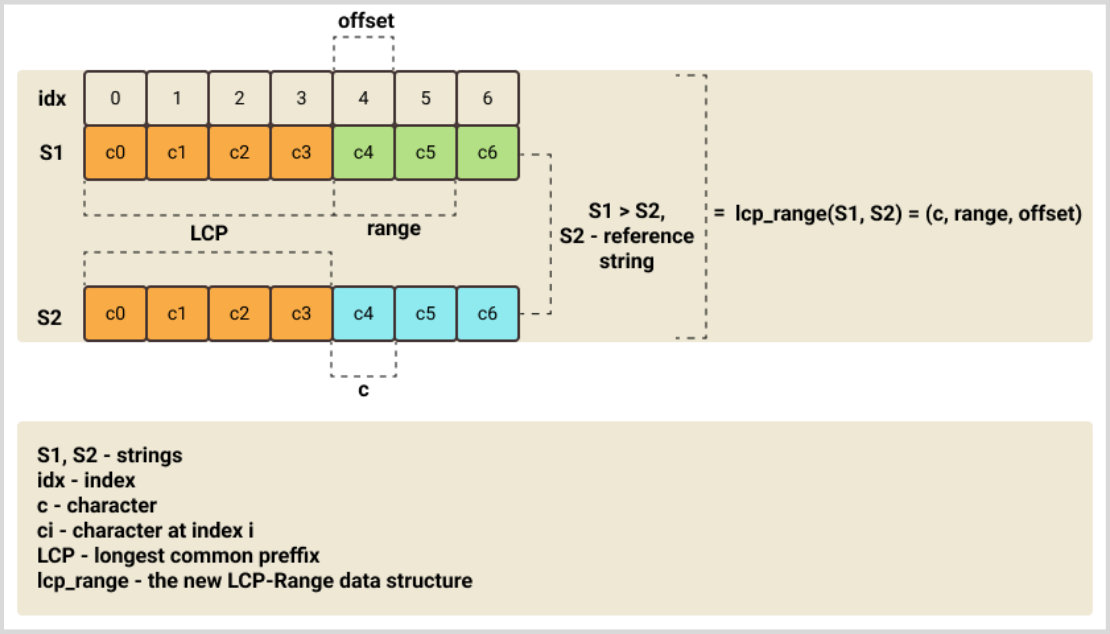

Figure 8: The LCP-Range data structure

For a better understanding, consider the following example illustrated below in *Figure 9: The LCP-Range data structure for string S1=abcdeef and string S2=abcdffj*. Suppose there are two strings *s1* and *s2* each consisting of the following character sequences: *s1 = abcdeef* and *s2 = abcdffj*. Among the two strings, *s1* is smaller than *s2*. Next, let us determine the longest common prefix (LCP) of *s1* and *s2*, which is *LCP = abcd*. This means that the offset where the two compared strings (*s1* and *s2*) start to differ will be the index of the first character after the LCP, which is actually the length of LCP: *length of LCP = length of abcd = 4 (characters)*. Therefore, the offset equals to 4 in the given example. The character at the offset index 4 in the smaller string *s1* is *“e”*. Finally, the range length is a constant defined beforehand. In this example, let the range length be equal to 2; then the

character sequence in the larger string *s2* starting at the offset index 4 and ranging 2 characters in length is: *"ff"*. Therefore, the LCP-Range data structure for *s1* and *s2* is calculated as the following triplet: *lcp_range(s1, s2) = (e, ff, 4)*. Here, the smaller string of the two is also referred to as the *reference string*. Therefore, in the given example, the reference string is *s1* since *s1* is smaller than *s2*.

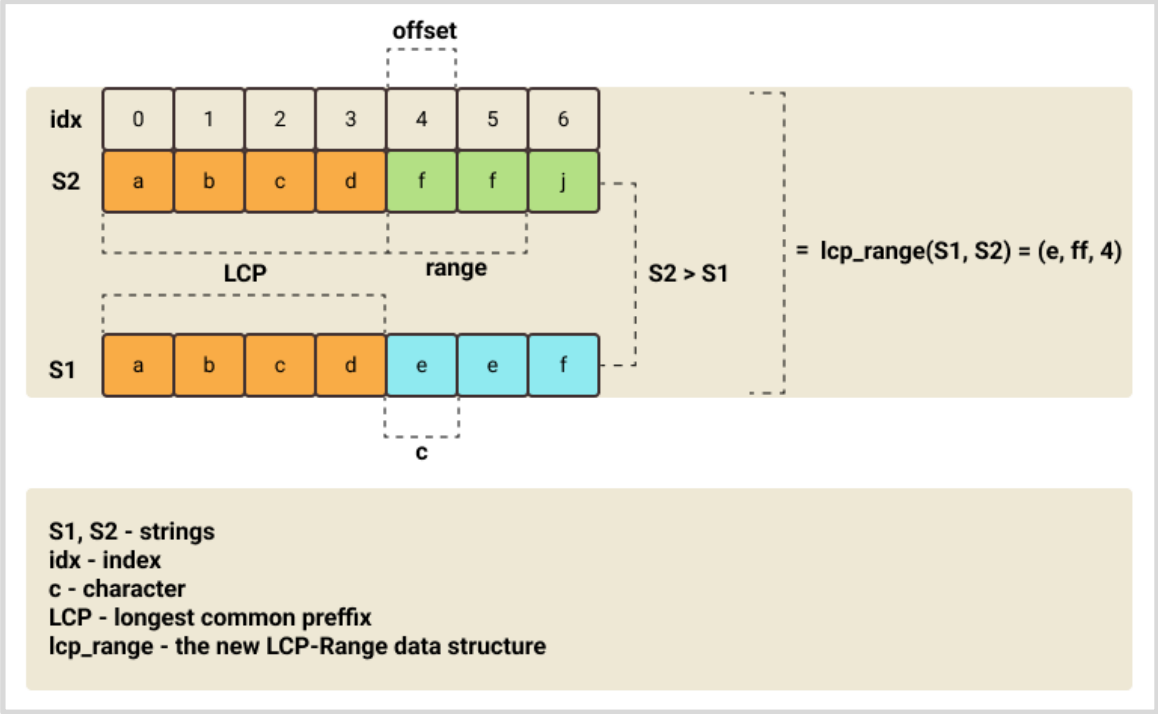

Figure 9: The LCP-Range data structure for string S1=abcdeef and string S2=abcdffj

Consequently, the LCP-Range array is an array that stores the LCP-Range data structures - more specifically, the LCP-Range information of suffix pairs that are following each other continuously in a lexicographically sorted suffix array. For instance, for the following suffix sequence - *suffix1, suffix2, suffix3, suffix4, suffix5* - the LCP-Range information will be calculated for each of the following pairs - *<suffix1, suffix2>, <suffix2, suffix3>, <suffix3,*

suffix4>, <suffix4, suffix5> - so, the resulting LCP-Range array will consist of the following LCP-Range data structures: *lcp_range(suffix1, suffix2), lcp_range(suffix2, suffix3), lcp_range(suffix3, suffix4), lcp_range(suffix4, suffix5)* [9,19].

Considering that the most diminutive suffix (the smallest suffix) does not have a reference string, the first element of the LCP-Range [9,10,11,12,13,18,19] array was replaced with the initial character range of the most diminutive suffix. Next, the LCP [9,19] between successive suffixes was calculated. As previously mentioned, all LCP arrays [9] with the same S-prefix that were generated after the local suffix sorting step were merged together resulting in a globally ordered LCP array [9]. The merging process was implemented based on LCP-Merge [9,19], using a multi-sort sorting tree - the loser tree [9] which is a type of tournament tree [9]. The figure below *Figure 10: An example of a 4-way loser tree* provides an example of the loser tournament tree:

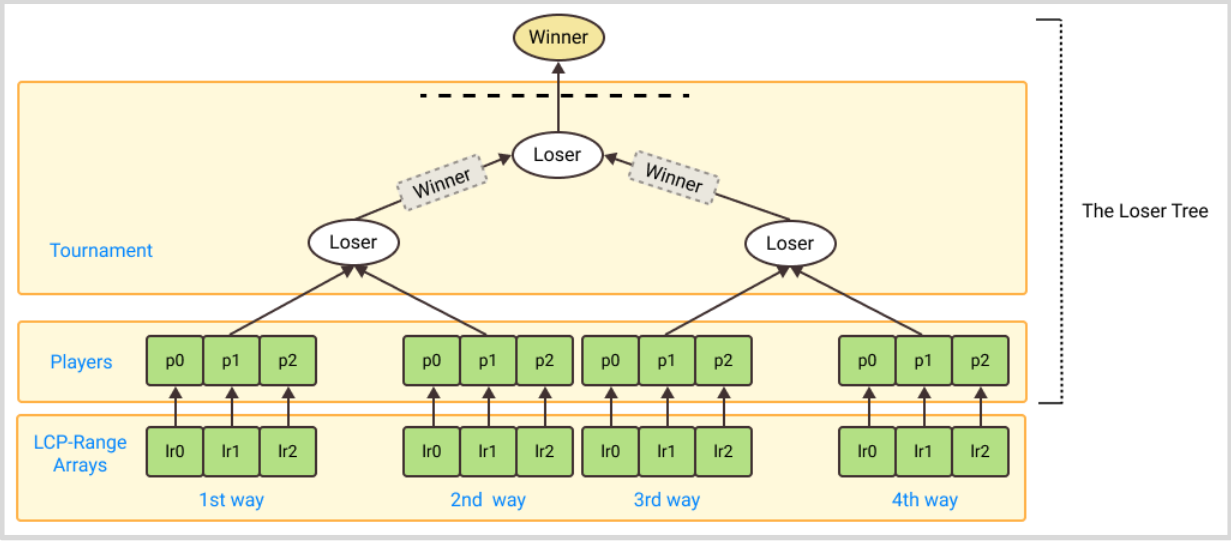

Figure 10: An example of a 4-way loser tree

It is important to note that the loser tree [9] has a remarkable property - for each player from the same path, there is a fixed direct path moving in the direction of the root node [9]. In particular, suppose that the final winner of the tournament is path w, then the next player p from where w enters gets to the top three losers. The player p is first compared with the LCP range information stored in its parent node. The two LCP ranges have the same reference string, which is the final winner of the last tournament. After comparing the LCP ranges, the winner moves to the next parent node. The winner information and the LCP range stored in the next parent node still have the same reference string, so the LCP-Range comparisons may be continued further on until the final winner is determined, so the rest of the process is similar: every two LCP ranges that are compared contain the same reference string [9,10,11,12,13,18,19]. Finally, the suffix subtree is built.

3.4 New Algorithm Complexity Calculation

Having implemented a new distributed optimized version of the ERa generalized suffix tree construction algorithm [1], next the new algorithm's complexity was calculated and the corresponding calculations are provided in the following paragraphs. Considering that the modeled algorithm consists of two main stages - the parallel subtree partitioning stage and the parallel subtree construction stage (parallel LCP array construction), the complexity of each stage was calculated and is provided below.

The complexity of the parallel subtree partitioning stage can be calculated based on the maximum length of the S-prefixes with frequencies below the FM frequency threshold, the complexity of building the frequency trie for each input partition, and the total number of iterations that took place before there were no more S-prefixes with frequencies exceeding the FM frequency threshold. Let *maxPrefixLen* denote the maximum length of the S-prefixes with frequencies below the FM frequency threshold. On average, *maxPrefixLen* takes up: $O(\log_{alphabetSize} \frac{totalSizeOfInputString}{FM})$ and, in the worst case scenario, the *maxPrefixLen* takes up at most: $O(totalSizeOfInputString)$ Next, considering that the input string S is divided into p partitions, the complexity of building the frequency trie for each input partition can be calculated as: $O(\frac{totalSizeOfInputString}{p} \times w)$, where p denotes the number of split partitions; and w denotes the count window for each iteration during the subtree partitioning step. In addition, the last factor, the total number of iterations, can be calculated as: $O(\frac{maxPrefixLen}{wstepSize})$, where wstepSize denotes the count window's step size. Therefore, combining each

factor's calculation together, the overall complexity of the parallel subtree partitioning stage turns out to be: $O(\frac{totalSizeOfInputString}{p} \times \log_{alphabetSize} \frac{totalSizeOfInputString}{FM}) \times p)$ However, since in practice *maxPrefixLen* is very small, it is not significant and can be disregarded, therefore, the overall complexity actually add up to $O(n)$.

The complexity of the parallel subtree construction stage can be presented in two parts: the local sorting step's complexity and the multi-way LCP-Merge sorting step's complexity. The local sorting step's complexity depends on the total number of suffixes that share the same starting S-prefix. This total number in the worst case would be $O(totalSizeOfInputString)$, nevertheless, since subtrees are partitioned beforehand, the total number is reduced and can be at most equal to the FM frequency threshold. The local sorting step's complexity for an S-prefix can be calculated as: $O(\frac{FM}{p} \times \log_2 \frac{FM}{p} \times p)$, where p denotes the number of split partitions. Finally, the multi-way LCP-Merge sorting step's complexity can be calculated as: $O(FM \times \log_2 k)$*[9]*, where k is the total number of rounds in the parallel subtree construction step.

3.5 Analyzing Performance Test Results

This section provides performance evaluation test results of the modeled and implemented algorithm for constructing generalized suffix trees on distributed platforms. In this work's implementation, Apache Hadoop [7] was used as the virtual distributed file system, an overview of the Hadoop distributed filesystem [7] is provided in *Figure 11: The overview of the Hadoop Distributed File System (HDFS)* below. At the time of writing, we conducted experiments on a local machine setup running on the Ubuntu 20.04 operating system, the Apache Spark [8] version that was used is 1.6.3, and Apache Hadoop [7] 2.6.5 was used. In the future, the plan is to further set up a physical cluster and run tests on larger data volumes to determine the areas that should be further optimized as the load increases.

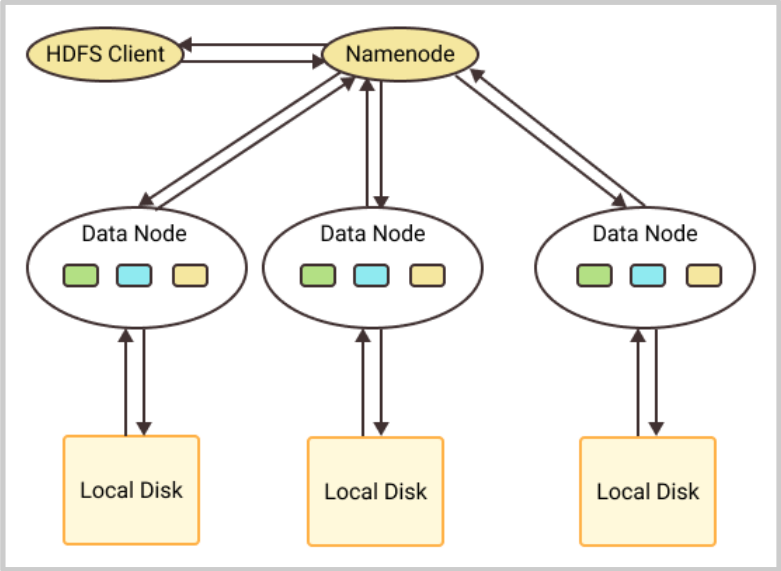

Figure 11: The overview of the Hadoop Distributed File System (HDFS)

To compare the performance gain of the implemented algorithm as opposed to the original state-of-the-art ERa [1] algorithm, the ERa [1]

algorithm was also executed on the same machine setup in the same environment to ensure that both implementations were compared in a fair and correct manner. Furthermore, both the optimized algorithm and the original ERa [1] implementation were given the same data set as the input. In this work, open public data sets were used taken from the Kaggle [23] platform, placed in the Hadoop distributed file system [7], and provided to both of the compared algorithm implementations.

During the whole process, execution statistics were output to the console and saved to output files to be analyzed further, such as the time that a certain process took during the three main stages of the suffix tree construction algorithm: the data preparation stage, the subtree partitioning stage, and the subtree construction stage. The code snippet below provides an example of such statistics:

```
[Stage 1] ====================> time:
Merge to File, time:423ms
repartition, time:280ms
Iter 1 start, currentLength = 2
counting: 23ms
File ends
[Stage 2] ====================> time:
collect: 6051ms
Terminal Num for length 2:2
prefix Num for length 2:70748
filter: 5ms
divide, time:6427ms
roots num:50 sum :70750
groupingMap, time:6ms
Grouping Num: 86 size:50
iters:1
```

```
17 iter 0: mean: 736  sigma:1305.0919507835454
18 ====================================
19 [Stage 3] Building group 1 start
20 ====================================
21 broadcast, time:4ms
22 LcpGen, time:16ms
23 groupByKey, time:10ms
24 subTreeMaterial, time:4ms
25 trieGraph, time:33ms
26 single Node LcpGen, time:9ms
27 …
28 firstMerge complete, time:2ms
29 ersort iteration complete, time:250ms
30 build suffix tree for root:C, num: 1235
31 …
32 Building group 1vcomplete. time: 17135ms
33 ====================================
34 my, time: 24359ms
35 ====================================
```

In the provided log fragment, on line #1 it can be seen that the first stage takes place - the data preparation step where the input string is partitioned and distributed among the processes, see section *3.1 New Algorithm Preparation Stage - Data Distribution* where this stage was described in detail. Next, on line #7 it is shown how the second stage is performed - the subtree partitioning stage, the implementation of which was described in detail in section *3.2 New Algorithm - Parallel Subtree Partitioning Stage* of this work. Finally, on line #19 the final subtree construction stage statistics are logged to the output, see section *3.3 New Algorithm - Parallel Subtree Construction Stage* for a detailed description of the subtree construction stage's implementation.

The Figure below *Figure 12: Graph comparison of the optimized and original ERa algorithm* provides a graph representation comparison of both the optimized and original algorithm. From the compared execution time statistics it can be concluded that the new approach for distributing generalized suffix tree construction is efficient and did indeed provide a performance gain of roughly 2-2.5 times faster than the aforementioned ERa [1] algorithm.

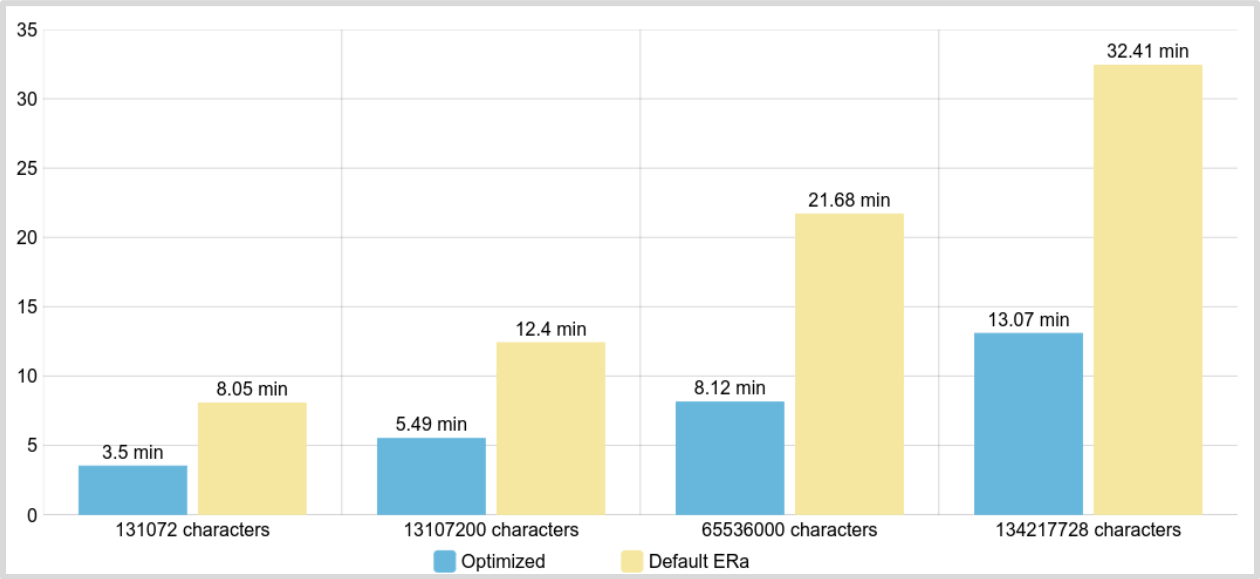

Figure 12: Graph comparison of the optimized and original ERa algorithm

Therefore, it can be stated that the optimization techniques applied in the algorithm that was modeled in this work for the suffix tree construction process on distributed platforms did improve the overall performance. As mentioned in section 2.2 The Elastic Range (ERa) Algorithm For Suffix Tree Construction, one of the performance gain causes is that the frequency counting algorithm used in ERa [1] for subtree partitioning is sequential. Furthermore, the count window is only extended by one, resulting in many

iterations and high I/O costs. On the other hand, in this work, the proposed algorithm uses a parallel frequency counting method to accelerate the subtree partitioning stage. Furthermore, the step size of the count window is increased by a larger number, so the number of iterations is reduced. As a result, the proposed algorithm outperforms ERa [1] at the subtree partitioning stage and provides better results in the overall performance measurements. In future work, the plan is to improve the proposed algorithm even further and achieve a greater performance gain by using in-memory distributed storage.

CONCLUSION

This work was inspired by the desire to enhance suffix tree construction when working with large alphabets and very long strings that exceed the available memory capacity. As a result, a highly scalable algorithm for large-scale generalized suffix tree construction on distributed parallel platforms was modeled. The entire process was presented in three main steps: the data preparation step, where the input string was partitioned, and the partitions were distributed among computing nodes; the subtree partitioning step; and, finally, the subtree construction step.

To partition the subtree and construct the subtree in the data parallel paradigm, a data-sharing strategy was applied. Then an efficient algorithm for subtree partitioning based on parallel frequency counting was proposed. Next, we looked at an effective strategy for allocating subtree construction tasks that can reduce reading costs and achieve load balance on individual nodes. Finally, during the subtree construction phase, we used the LCP-Range [9,10,11,12,13] data structure and the multi-faceted LCP-Merge [9,10,11,12,13] sorting algorithm to build the LCP array [9,10,11,12,13] in a parallel manner.

Experimental results on Apache Spark [8] revealed that the modeled algorithm significantly outperforms the state-of-the-art ERa [1] algorithm by about 2-2.5 times acceleration difference; therefore, the resulting algorithm for distributed generalized suffix tree construction is quite appropriate for applications that wish to improve large text data operations and that are

based on a distributed system. In addition, the modeled algorithm can efficiently construct a generalized suffix tree for a set of strings.

REFERENCES

1. Essam Mansour, Amin Allam, Spiros Skiadopoulos, ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings, Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 1, pp. 49-60 (2011)
2. Martin Kleppmann Designing Data-Intensive Applications. O'Reilly Media, Inc., March 2017. ISBN: 9781449373320
3. M. Comin, M. Farreras, Efficient parallel construction of suffix trees for genomes larger than main memory, in: Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI), ACM, 2013, pp. 211–216
4. Tshepo Kitso Gobonamang, Dimane Mpoeleng,Counter based suffix tree for DNA pattern repeats,Theoretical Computer Science,Volume 814,2020,Pages 1-12,ISSN, 0304-3975,https://doi.org/10.1016/j.tcs.2019.12.014 (https://www.sciencedirect.com/science/article/pii/S0304397519307893)
5. Ghoting A., Makarychev K. (2011) Suffix Trees. In: Padua D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-09766-4_464
6. Computing and Combinatorics: 8th Annual International Conference, COCOON 2002, Singapore, August 15-17, 2002 Proceedings, Ibarra, O.H. Zhang L., https://books.google.com.ua/books?id=3XRrCQAAQBAJ
7. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
8. https://spark.apache.org/

9. Parallel Multiway LCP-Mergesort, A Eberle, Andreas, 2014, english, Eberle2014_1000042457, 10.5445/IR/1000042457, http://dx.doi.org/10.5445/IR/1000042457
10. https://www.cs.helsinki.fi/u/tpkarkka/opetus/11s/spa/lecture10.pdf
11. Ng, Waihong & Kakehi, Katsuhiko. (2008). Merging String Sequences by Longest Common Prefixes. Ipsj Digital Courier. 4. 69-78. 10.2197/ipsjdc.4.69.
12. Bingmann, T., Eberle, A., and Sanders, P., “Engineering Parallel String Sorting”, 2014.
13. Barsky, Marina & Stege, Ulrike & Thomo, Alex. (2010). A survey of practical algorithms for suffix tree construction in external memory. Software: Practice and Experience. 40. 965 - 988. 10.1002/spe.960.
14. E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260
15. Zaki, Mohammed J. and Benjarath Phoophakdee. “Trellis: genome-scale disk-based suffix tree indexing algorithm.” (2007).
16. Matteo Comin and Montse Farreras. 2013. Efficient parallel construction of suffix trees for genomes larger than main memory. In Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI '13). Association for Computing Machinery, New York, NY, USA, 211–216. https://doi.org/10.1145/2488551.2488579
17. https://www.sciencedirect.com/topics/agricultural-and-biological-sciences/bernoulli-distribution
18. P. Flick, S. Aluru, Parallel construction of suffix trees and the all-nearestsmaller-values problem, in: Proceedings of the 31st IEEE

International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp.12–21, doi: 10.1109/IPDPS.2017.62

19. Guanghui Zhu, Chen Guo, Le Lu, Zhi Huang, Chunfeng Yuan, Rong Gu,Yihua Huang, DGST: Efficient and scalable suffix tree construction on distributed data-parallel platforms, Parallel Computing
20. https://developers.google.com/optimization/bin/bin_packing
21. Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. 2018. Optimal Multi-Way Number Partitioning. J. ACM 65, 4, Article 24 (August 2018), 61 pages. https://doi.org/10.1145/3184400
22. Richard E. Korf, A complete anytime algorithm for number partitioning, Artificial Intelligence, Volume 106, Issue 2,1998, Pages 181-203, ISSN 0004-3702, https://doi.org/10.1016/S0004-3702(98)00086-1. (https://www.sciencedirect.com/science/article/pii/S0004370298000861)
23. https://www.kaggle.com/datasets