Modeling Distributed Generalized Suffix Trees For Quick Data Access

Written by Vera Didenko

Supervised by Andriy Glybovets

The problem at hand

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.

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.

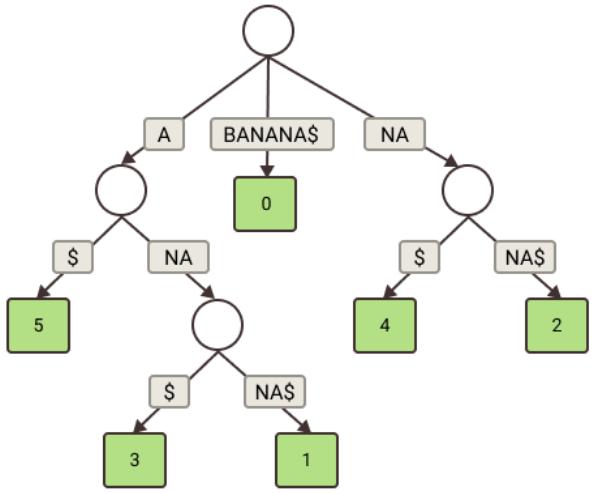
The process of building a suffix tree for massive data sets is a very time-consuming task.

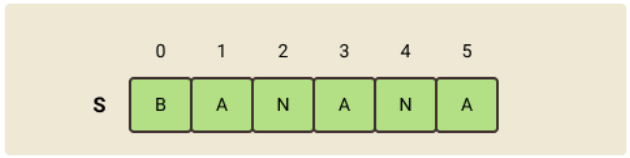
Goal

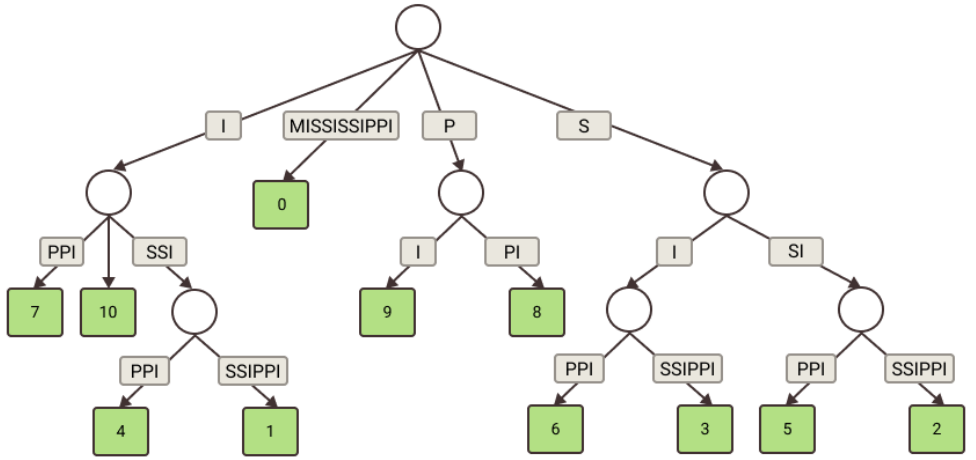
The aim of this work is to distribute generalized suffix tree construction, so the process is efficient in terms of time complexity and memory consumption.

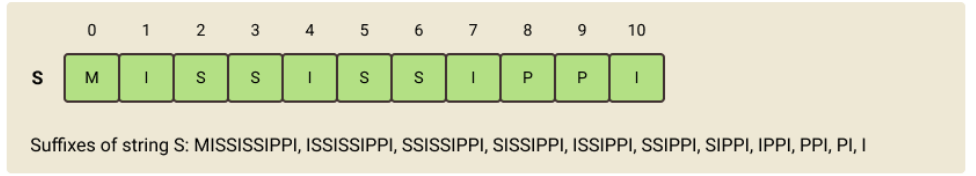
A distributed approach to constructing the suffix tree will allow working with large alphabets and very long strings that exceed the available memory capacity.

In this work, an efficient and highly scalable algorithm for constructing generalized suffix trees on distributed parallel platforms was modeled by optimizing ERa.







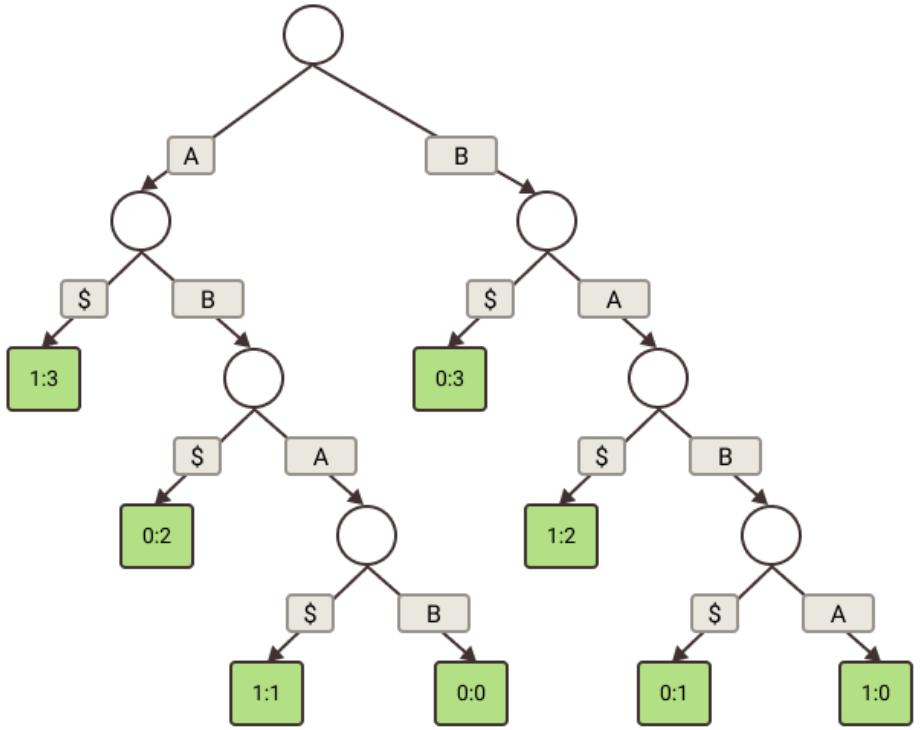


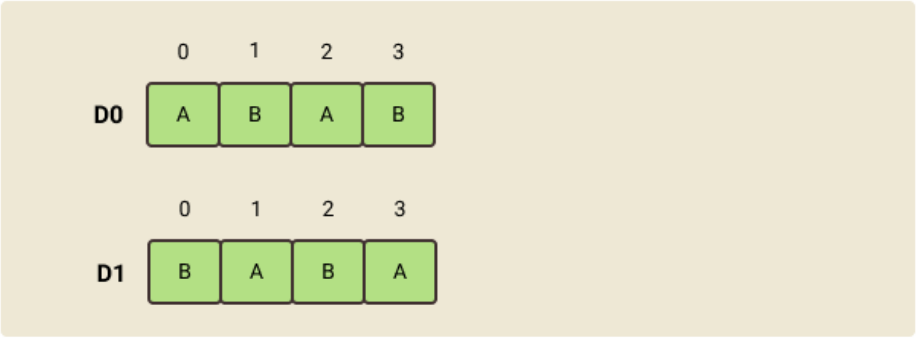
Background on suffix trees

- The suffix tree 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.

Generalized suffix trees

- A generalized suffix tree 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 work, 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 - the start and end position of each string in the merged string, can be used to distinguish between different strings in the merged string.





New algorithm by optimizing ERa

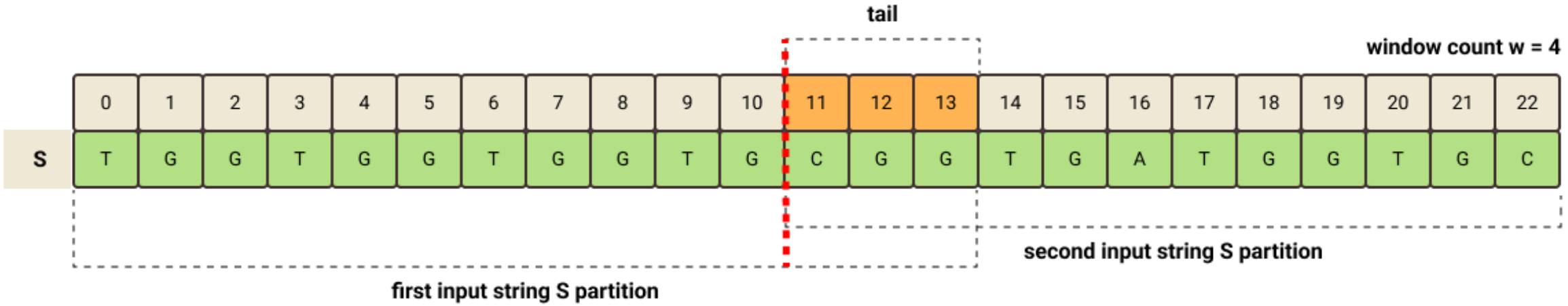
Preparation step – data partitioning stage

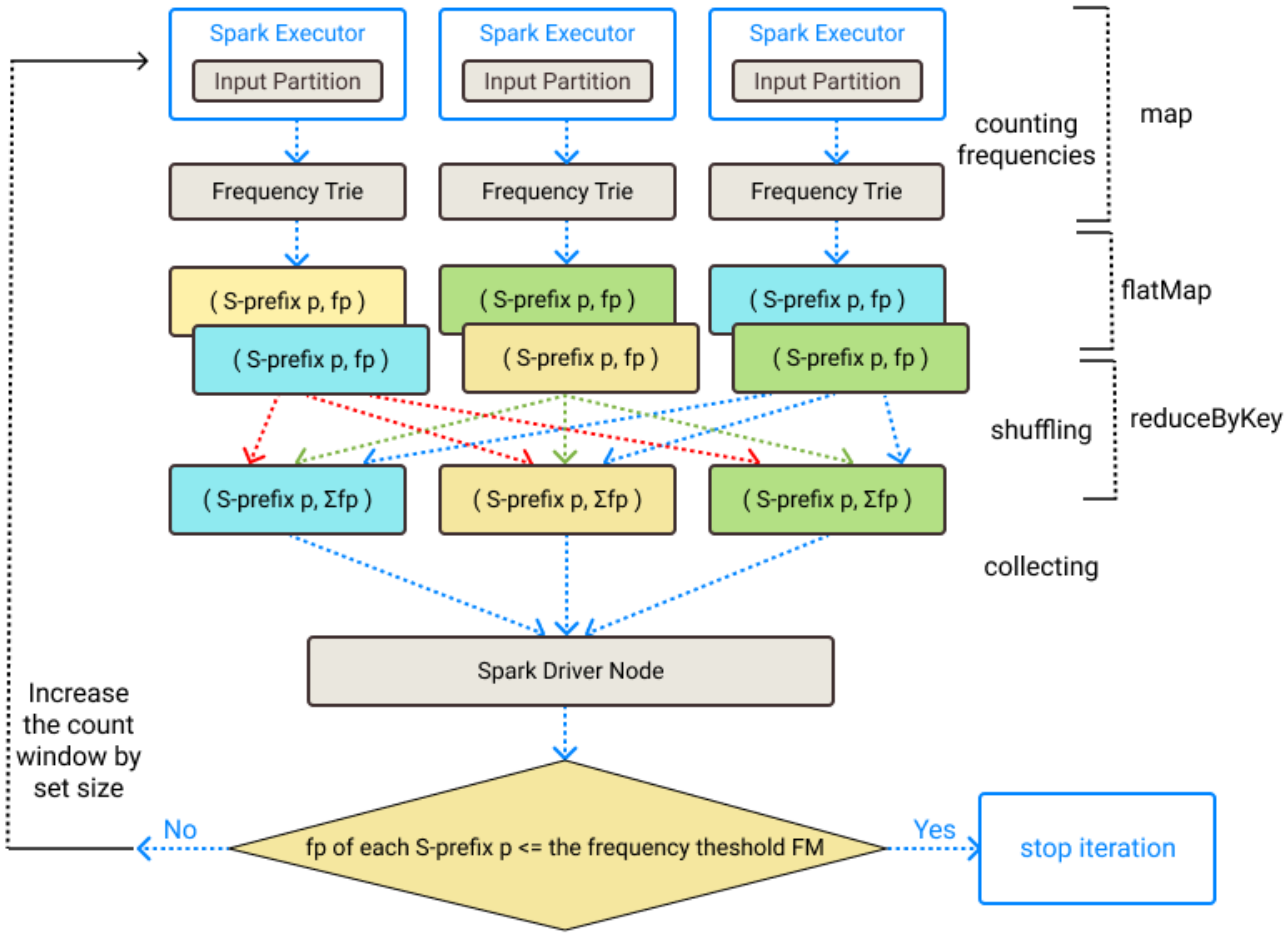
Parallel subtree partitioning stage

Parallel subtree construction stage

Data partitioning stage

- 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 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.



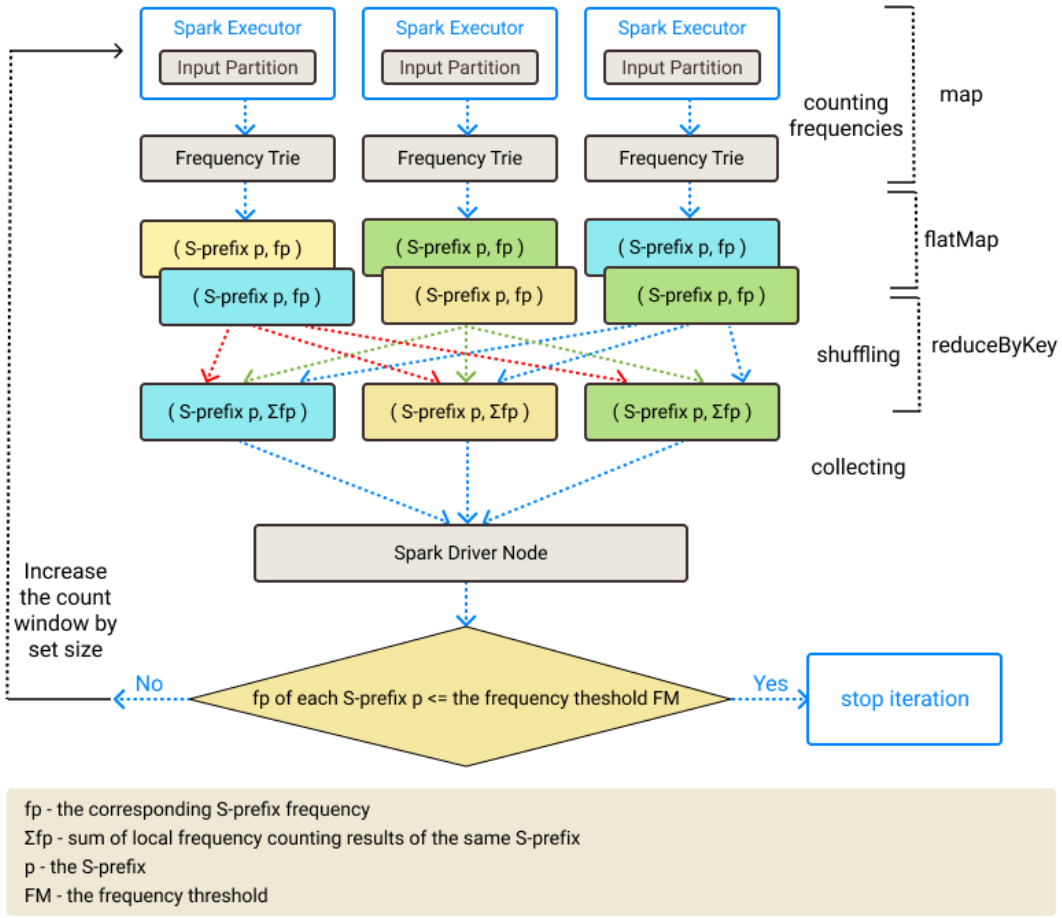


fp - the corresponding S-prefix frequency
Σfp - sum of local frequency counting results of the same S-prefix
p - the S-prefix
FM - the frequency threshold

Parallel subtree partitioning stage

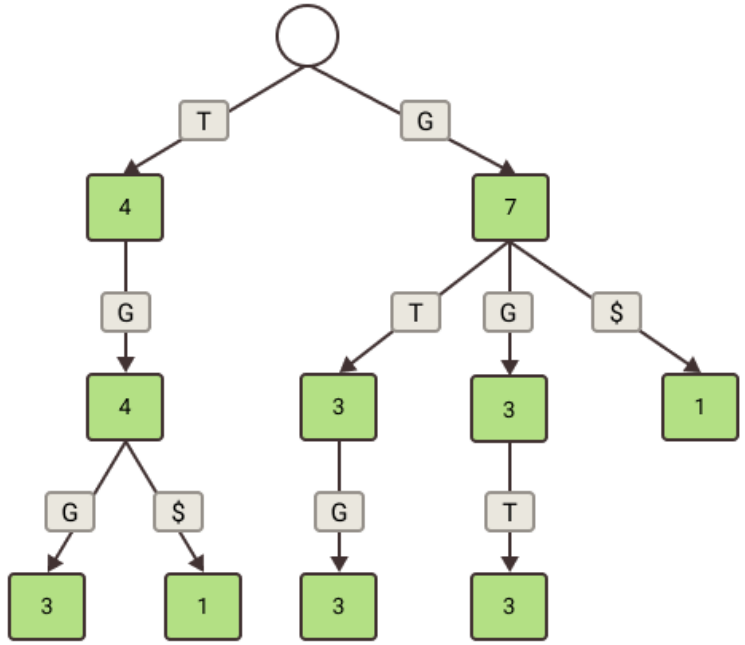
- Calculating the occurrence frequency for the S-prefix
- The frequency trie
- The incremental window count approach

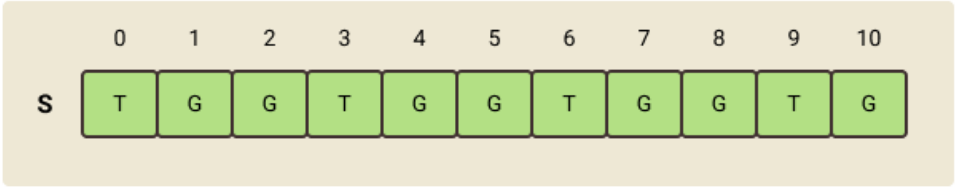
Parallel subtree partitioning stage – S-prefix frequency calculation



- 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.

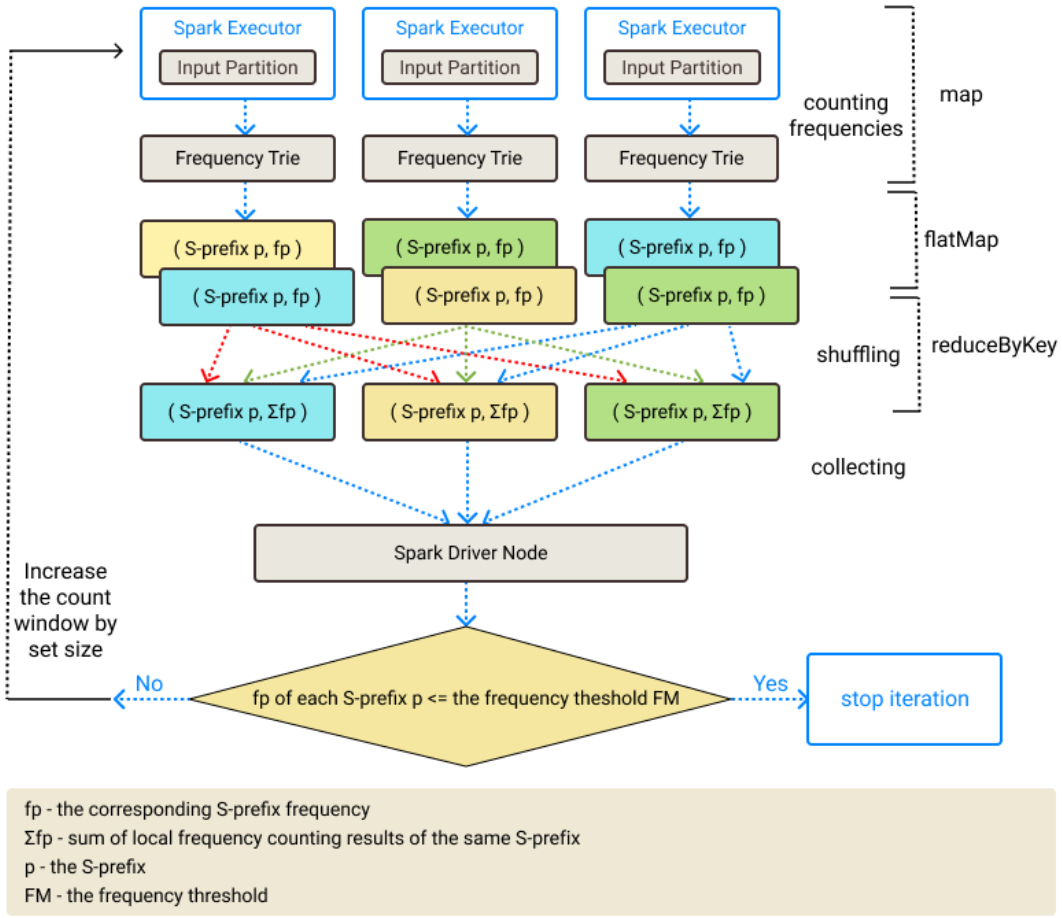
Parallel subtree partitioning stage – frequency trie



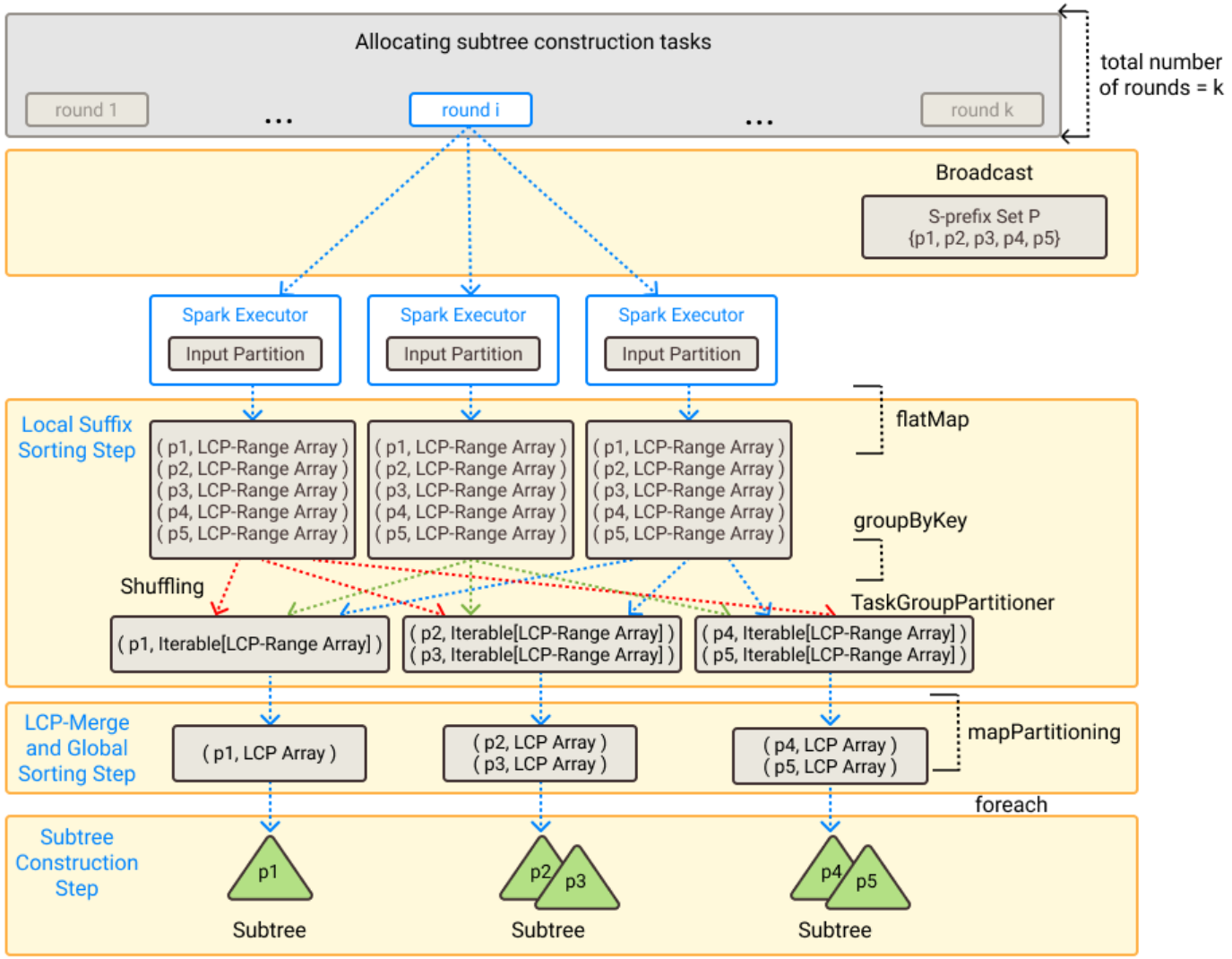


- Using a frequency trie, the total number of iterations (that is, Input/Output accesses) was reduced by calculating the frequencies of different lengthed S-prefixes by simply scanning the input string.

Parallel subtree partitioning stage – incremental window count

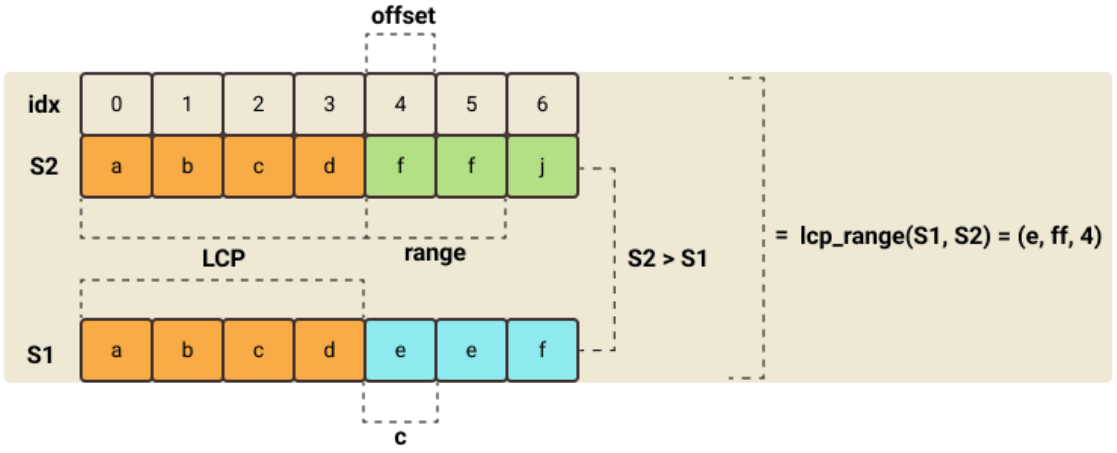


- The general process of subtree partitioning in a parallel manner is organized in multiple iterations, during which the count window increases by the defined step size.
- Let the initial count window be winit, then during the first iteration, the frequencies of all winit lengthed S-prefixes are counted.
- In the next iteration, the window size is increased by the defined step size. The frequencies of all S-prefixes that have a length that fits into the [winit, wnext] interval are calculated in this iteration.

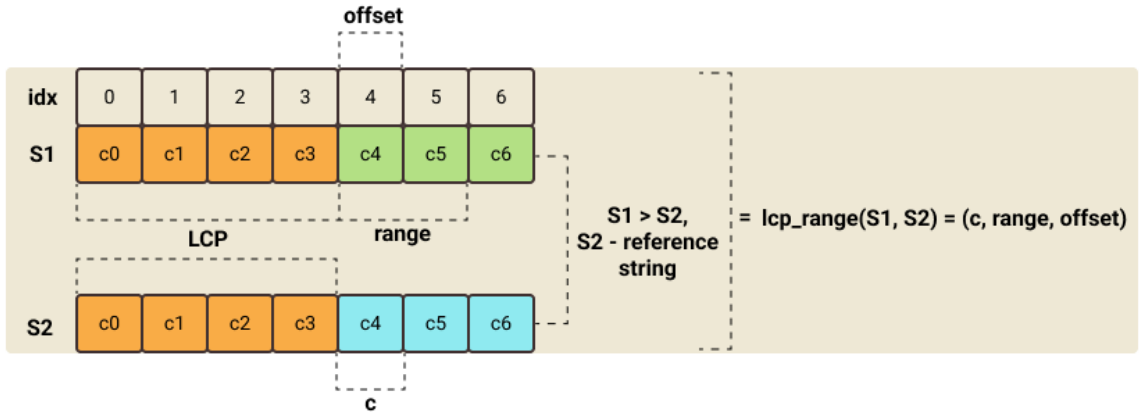


Parallel subtree construction stage

- The distribution of subtree construction tasks is affected by two main factors: the cost of input/output calls and keeping the worker nodes balanced.
- We used the Bin-Packing and the Number-Partitioning algorithms in combination to reduce the I/O cost and ensure node balance.



S1, S2 - strings
idx - index
c - character
LCP - longest common preffix
lcp_range - the new LCP-Range data structure



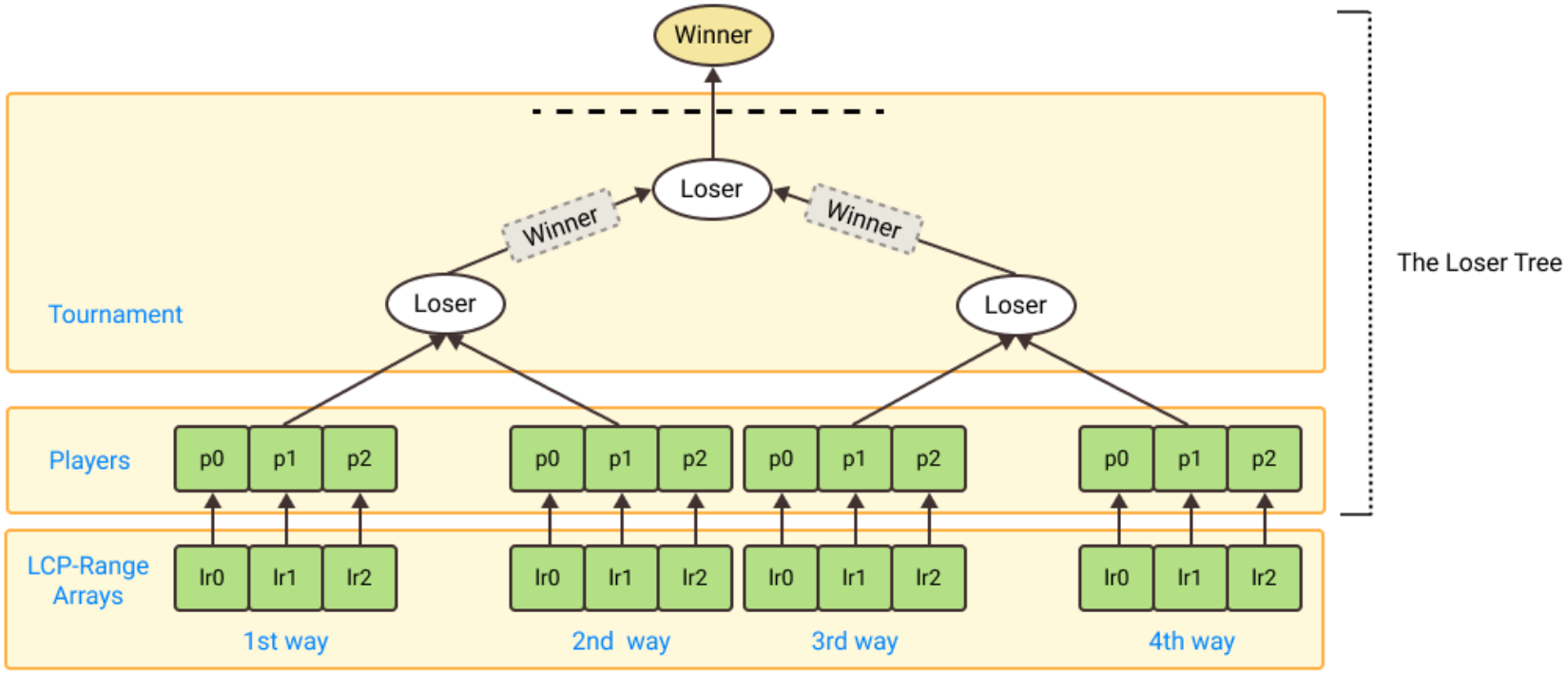
S1, S2 - strings
idx - index
c - character
ci - character at index i
LCP - longest common preffix
lcp_range - the new LCP-Range data structure

Parallel subtree construction stage

- We created an LCP-Range data structure for a more efficient way to compare the suffixes.
- The LCP-Range array contains LCP-Range information for each pair of subsequent suffixes in lexicographical order.

Parallel subtree construction stage

- Also, we utilized the loser tournament tree data structure in the LCP-Merge Sorting step.
- As a result, a global LCP-Array is generated.
- Finally, the subtree is constructed (recording of the algorithm execution).



Calculating algorithm complexity

THE COMPLEXITY OF THE PARALLEL SUBTREE PARTITIONING STAGE

THE COMPLEXITY OF THE PARALLEL SUBTREE CONSTRUCTION STAGE

Calculating algorithm complexity - the parallel subtree partitioning stage

- The complexity of the parallel subtree partitioning stage:

$$O\left(\left(\frac{totalSizeOfInputStringS}{numberOfPartitions}\right) \cdot log_{(alphabetSize)}\left(\frac{totalSizeOfInputStringS}{frequencyThresholdFM}\right) \cdot numberOfSparkExecutors\right)$$

Calculating algorithm complexity - the parallel subtree construction stage

The local sorting step's complexity for an S-prefix:

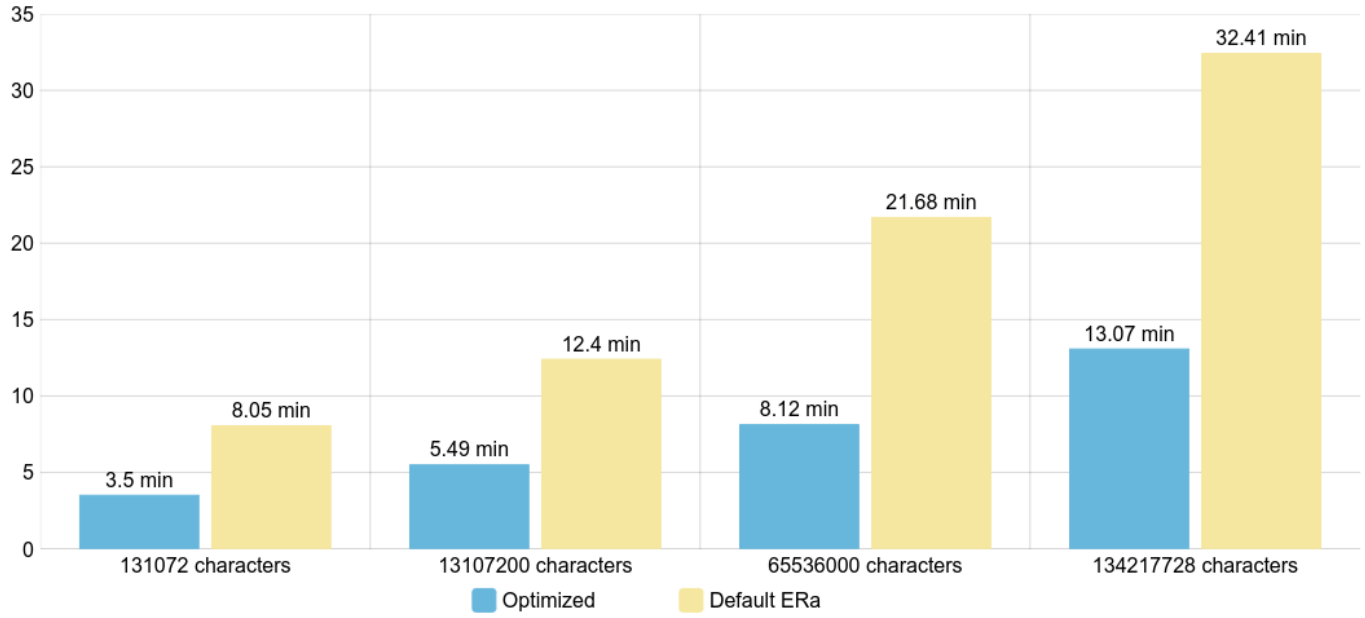
$$O\left(\left(\frac{frequencyThresholdFM}{numberOfPartitions}\right) \cdot log_2\left(\frac{frequencyThresholdFM}{numberOfPartitions}\right) \cdot numberOfSparkExecutors\right)$$

The multi-way LCP-Merge sorting step's complexity:

$$O\left(frequencyThresholdFM \cdot log_2\left(totalNumberOfRounds\right)\right)$$

Performance comparison

- Local machine setup, Ubuntu 20.04
- Apache Spark 1.6.3
- Apache Hadoop 2.6.5 was used for the distributed file system
- Both algorithms were executed on the same machine setup in the same environment and given the same input data to ensure that both implementations were compared in a fair and correct manner.
- Performance gain by roughly 2-2.5 times



Conclusion

Experimental results on Apache Spark revealed that the modeled algorithm significantly outperforms the state-of-the-art ERa algorithm by about 2-2.5 times

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.

Thank you for your attention!