I J C T A, 9(41) 2016, pp. 1209-1217

Map Reduce Based Probabilistic Generalized Suffix Tree Construction

Vishnu Shankar Tiwari[*] and Arti Arya[**]

ABSTRACT

String sequence indexing is thebasis of many applications including Route Prediction, Bio-Informatics, Text processing, String matching etc. where thegoal is to index a huge sequences. Moreover, prediction application requires probabilistic suffix tree. Probabilistic Suffix Tree (PST) is one of widely used string sequence indexing technique and also serves amodel for prediction. Many available sequence indexing uses suffix tree which is built over one long sequence.The generalized suffix tree is essentially a suffix tree built from a huge number of smaller sequences. PST construction from thehuge volume of data by processing sequentially is a bottleneck in the practical realization. Most of theexisting works focused on time-space tradeoffs on asingle machine. Extending them to distributed systems is still challenging. Attempts are made to achieve parallelism and scalability by chopping long single input string into smaller sequences and processing them in parallel. Then merging smaller subtrees to produce final suffix tree. This approach leads to huge intermediate subtrees. Proposed technique leverages map-reduce computation framework for distributed construction of suffix tree. Huge collection of smaller string sequences is processed to construct suffix tree. Collection of intermediate subtrees generatedare made proportional to number of computing nodes in thecluster and independent of input string size. The probabilistic aspect of thetree is also taken care so that it can be used as amodel for prediction application.End product is probabilistic generalized suffix tree (PGST). Implementation is Hadoop based and provides distributed replicated storage of data with fault tolerance.

Keywords: Suffix Tree, Big Data, Map Reduce, Hadoop, HDFS

I. INTRODUCTION

Suffix tree is widely used in pattern recognition and machine learning [1]. Traditionally it is used in compression, text analytics,bio-informatics, genome sequence analysis, route prediction, speech and language modeling, text mining etc. [1] [2] [3]. Suffix trees facilitate improved performance of searching on the indexed string [5]. A compact TRIE data structured created from suffixesof a long sequence is known as suffix tree [2] [4]. Another variant created from a large number of sequences is known as generalized suffix tree [3]. Most of the existing work in this area focused on theefficient construction of Suffix tree on a single machine [6] [7] [8] [9] [10] [11]. PST construction process deals with huge data sets and processing such volume of data sequentially and coming up with a trie is a bottleneck in the practical realization.In most of existing techniques, scalability was achieved by leveraging multiprocessor systems and increasing internal memory[14] [15]. This comes under thecategory of vertical scalability and is limited by theamount of hardware a single system can support. Alternative is distributed computing based horizontal scalability where theprocess runs on distributed commodity hardware. Recent attempts in [4] [5] [18] tried achieving parallelism by chopping input string into smaller units and processing them in parallel to produce intermediate trees on disk. Finally merging intermediate trees to get final suffix tree. This leads to a huge number of subtrees to merge [4]. Our approach generates set of suffixes asintermediate result. Number of such sets generated ae in order of a number of nodes participating in computation and independent of input size. Sometimes because of application's requirement and also to achieveparallelism suffix tree built from

[*] Indian Institute of Technology, Mumbai, India, *E-mail: vishnustiwari@gmail.com*

[**] Department of Computer Application, PESSE, Visvesvarya Technological University, Bengaluru, India, *E-mail: artiarya@pes.edu*

acollection of sequences rather than one single sequence [3]. We present distributed processing technique to construct generalized suffix tree from a large number of relatively smaller sequences. The proposed technique is a two-step process. In thefirst step, suffixes of the input sequences along with their frequency of occurrence are computed and in thesecond step, ageneralized suffix tree is computed. Proposed two-step process makes itintuitive for distributed map reduce based computation without losing accuracy and efficiency. In Map Reduce model- Mappers computes all the suffixes along with their frequency count and Reducer constructs final probabilistic generalized suffix tree (PGST). As part of processing frequency of occurrence of each suffix is taken care and probability of nodes of thetreeare computed. This is an essential requirement in many prediction applications which are interested not only in suffix tree butalso theprobability of occurrence of anode of thetree. Rest of the paper is arranged as follows. Existing work on suffix tree and contribution of this work is discussed in section II. Section III describes Suffix trees and generalized suffix tree. A two-step process for constructing generalized suffix tree is presented in section IV which can be used to run on a single machine. Approach discussed in intuitive for distributed computation.Map-reduce based horizontally scalable process of constructing PGST is presented in section V. Performance evaluation results are discussed in section VI.

II. RELATED WORK

Suffix tree was first introduced by Weiner [13]. It was further simplified by McCreight [6] and Ukkonen [7] and optimal construction is proposed. Naïve approach can take up to $O(n^3)$ time but Ukkonen [7] can build suffix tree in linear time $O(n)$. But they work on a one long string and there is no way can be made parallel. Also they are in-memory based and suffers with memory constraints [2] and suffers with memory locality of reference [4]. Researches tried removing bottleneck making it disk based. Hunt's algorithm [9], TDD [16], ST-Merge [17] and TRELLIS [12] are in this category but solves issue partially [4]. They comes under category of semi disk based. These approaches partitions long string into multiple subtrees and writes to disk and then merges them. Complexity of building tree is reported to be $O(n^2)$. None of them are parallel and distributed and hence merging phase requires lots of inter-processor communication. If everything fits into memory then they perform better than Ukkonen's algorithm but as soon as it goes beyond memory capacity they become inefficient. Scalability is severe issue with them. Recent two methods proposed wavefront [18] and B^2ST [19] actually made it possible to compute even when input size is larger than memory available. B^2ST [19] uses suffix arrays instead of partitioning suffix tree.A suffix array is an array containing all suffixes of the input sorted in lexographic order. Long input string is chopped in smaller units and builds suffix arrays in memory and dumps onto the disk. In next phase merging is performed. Focus is on reducing I/O but parallelism is not addressed. It also takes complexity of $O(n^2)$. Wavefront [18] focused on parallelism. Unlike others this works on whole string and partition is created on suffixes which shares common S-prefixes [4] [20]. Suffix subtree are constructed for each partition and then merged in last phase. Requires all the time two buffers- one the tree is getting merged which consumes around 50% of memory and another is resultant tree. Focus is mainly on leveraging multi core architecture. It is implemented and tested on IBM BlueGene/L super computer. However scalability cannot be achieved indefinitely because of tiling overhead [18]. ERA [4] took more intelligent approach which first chops the single long input sequence into smaller segments. And then subtrees are built for each independent partition. Then each subtree is divided horizontally and subtrees are merged to give final subtrees. It avoids multiple traversal of the subtreeto reduce I/O costs. Parallelism is tested in multi core system as well as distributed and reported complexity of $O(n^2)$. In order to achieve parallelism wherever applicable relies on chopping input string into smaller segments and construct subtrees for each of such segment. Authors in [5] proposed map reduce based horizontal scalable suffix construction based on Ukkonen's algorithm [7]. Input string is decomposed into smaller units following the same approach as in ERA [7]. Then for each partition Ukkonen's algorithm [7] for in memory computation of subtree. Hence drawback is if the partition is long enough to not fit in

memory technique becomes inefficient. Map Reduce is used for distributed computation. Mapper modules are used to compute subtree and dumps them on to disk. Hence output is a forest composed of smaller suffix trees. Output of this work is unusable as not one final tree is constructed. All other existing techniques generates final tree and is still a bottleneck.

Table 1
Comparison of the most important algorithms for suffix tree construction

	Complexity	*Memory Locality*	*Parallel*	*Probabilistic*
McCreight [6]	$O(n)$	Poor	No	No
Ukkonen [7]	$O(n)$	Poor	No	No
Hunt [9]	$O(n^2)$	Good	No	No
TRELLIS [12]	$O(n^2)$	Good	No	No
TDD [16]	$O(n^2)$	Good	No	No
ST-Merge [17]	$O(n^2)$	Good	No	No
Wavefront [18]	$O(n^2)$	Good	yes	No
B2ST [19]	$O(n^2)$	Good	No	No
ERA [4]	$O(n^2)$	Good	yes	No
Map-Reduce Ukkonen [5]	$O(n^2)$	Good	yes	No
Proposed PGST	$O(n^2)$	Good	yes	yes

Proposed work attempts to address shortcomings of the existing techniques. Focus is on distributed computation of suffix tree on acluster of independent machines. Map-reduce framework over Hadoop is used. One important issue tackled is partitioning. Even if theinput string is divided into multiple shorter chunks, resulting chunks are packaged in groups and dispatched to distributed node for computation and each such node will generate exactly one intermediate resultant suffix set. So thenumber of suffix sets generated independent of how many parts input string is broken but depends on how many nodes participated in computation which is constant for cluster. Since it works on theset of string we call it as ageneralized suffix tree. Without loss of generality, as others, in case input is one long string then the input string can be chopped into multiple smaller units and proposed technique can be used. Many prediction application does not require only suffix tree but also theprobability of occurrence of each node of the suffix tree. The author in [1] uses Suffix tree as Markov model for prediction in DNA sequence, music, text etc. for which probabilistic tree is essential. Other such example includes route prediction which requires Probabilistic Suffix Tree (PST). We address this essential requirement and keeps arecord of frequency of occurrence of each suffix in mapper module and at later stage passed toreducer module so that probability of each node in tree can be calculated. This tree is we call Probabilistic Generalized Suffix Tree (PGST).

III. SUFFIX TREE AND GENERALIZED SUFFIX TREE

Let $\Sigma = \{A, B, \ldots . Z\}$ denote a finite alphabet set of characters. Σ^* denotes the set of all finite length strings formed using characters from Σ. Let $X = x_0, x_1, \ldots ., x_{n-1}$ \$ with xi $\in$ & $X \in \Sigma^*$ denote an input string of length $n = |X|$ and \$ $\notin \Sigma$. Concatenation of two strings X and Y denoted as XY has length $|X| + |Y|$ and consists of alphabets from X followed by alphabets from Y such that $XX = x_0, x_1, \ldots ., x_{n-1}, y_0, y_1, \ldots . y_{m-1}$. A string Y, is prefix of another string X, denoted as $Y \ll X_i$, if $X = ZY$ for some string $Z \in \Sigma^*$. Similarly a string Y, is suffix of another string X, denoted as $Y \gg X_i$, if $X = ZY$ for some string $Z \in \Sigma^*$. For $X = ABCB$ all prefixes are ε, A, AB, ABC, $ABCB$ and all possible suffixes are ε, B, CB, BCB, $ABCB$. Empty string $\varepsilon \in \Sigma^*$ has length zero and is both prefix as well as suffix. Hence number of prefixes and suffixes of a string X is $|X|$. Given a string X all prefixes as well suffixes can be computed in time $\Theta|X|$ each.

The ordered arrangement of all $|X|$ suffixes of string X in a compact TRIE is known as the suffix tree T of X. For demonstration purpose let us assume alphabet set $\Sigma = \{A, C, G, T\}$ and a string $X = AATGG\$$. All the suffixes are as shown in Table 2 and resulting Suffix tree is as shown in Figure 1.

Table 2
All suffixes for AATGG$

i	x_i	Suffix
1	x_1	AATGG$
2	x_2	ATGG$
3	x_3	TGG$
4	x_4	GG$
5	x_5	G$
6	x_6	$

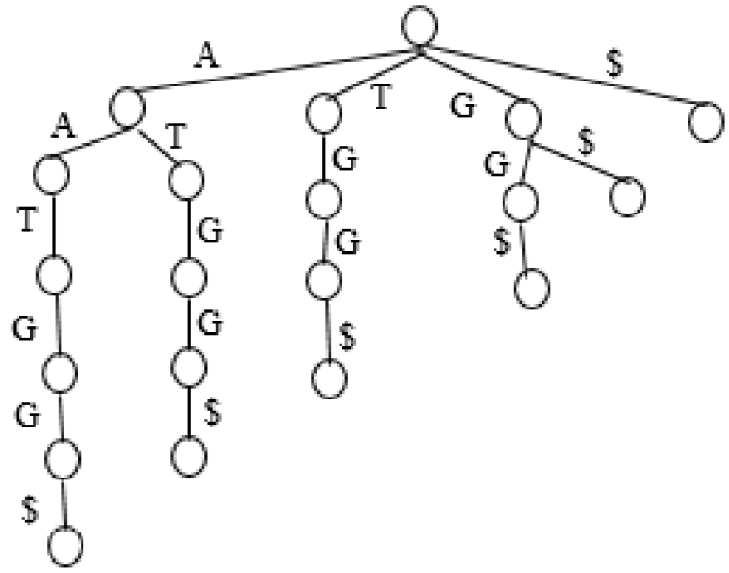

Figure 1: Suffix tree of string AATGC$

Some of thekey features of Suffix tree are listed below:

- Internal nodes can have max children equal to $|\Sigma|$. Hence branching factor of thetree is $|\Sigma|$. Leaves have zero children.
- Each edge in suffix tree is labeled by an alphabet $a \in \Sigma$.
- For any node, *path* (v) is the string formed by concating labels starting from root the the of the tree. If v is leaf node then *path* (v) corresponds to a suffix of the string.
- For each leaf u, the *path* (u) represents a suffix. Number of leaves is $|X|$ where X is string for tree construction.
- If two nodes u and v has same parent x then label $(x, u) \neq$ *label* (x, v) where (x, u), (x, v) are edges of tree. In other words no two edges out of a node can have edge labels.

Generalized suffix tree was initially proposed in [19]. Unlike other suffix trees which process one long sequence. The generalized suffix tree is constructed from a set of string. This work focuses on theconstruction of generalized suffix tree through distributed computing leveraging map-reduce processing framework. As part of processing frequency of occurrence of each suffix is taken care and probability of nodesare computed. This tree is known as Probabilistic Generalized Suffix Tree (PGST).

IV. PROBABILISTIC GENERALIZED SUFFIX TREE (PGST) CONSTRUCTION BASICS

A suffix tree constructed from a collection of strings is known as generalized suffix tree. Given is a set of strings $S = \{S_1, S_2 \ldots .. S_n\}$ where $S_i \in \Sigma \; \forall \; 1 \leq i \leq n$. Each S_i is appended with symbol $\$ \notin \Sigma$ so that none of suffix is a substring of any other suffix. This ensures each suffix creates a leaf in the tree.We propose two-step process to compute generalized suffix tree from S. First phase computes all suffixes and second phase constructs tree from suffixes computed earlier. Both steps are described below.

In the first phase, all the suffixes are generated and are put on a map which stores suffix as key and frequency as value. For each all suffixes are calculated and put in a map where key is suffix itself and value is number of time it occurs in D. Processing is as summarized in algorithm 1 below.

Algorithm I: Suffix frequency calculation

Input: Set of strings $S_1, S_2 \ldots . S_n$
Output: Map of all suffixes with frequency $m(suffix, count)$
Algorithm:
For each string$S_i \in S_1, S_2 \ldots . S_n$ repeat 1 and 2

I. Compute all suffixes of S_i
II. Insert suffixes calculated in 1 in a map. If suffix exists in map increment count Else insert suffix and set$count$ 1.

i	X_i	Suffix	Count
1	X_1	ACGT$	2
2	X_2	CGT$	2
3	X_3	GT$	4
4	X_4	T$	4
5	X_5	$	4
6	X_6	APGT$	1
7	X_7	PGT$	1

Table 3: Suffixes and their occurrence count

The process is demonstrated through the example below. Following four sequences we take for demonstration and is used as running example in all next sections. $S_0 = ACGT\$ \;\; S_1 = ACGT\$ \;\; S_2 = APGT\$ \;\; S_3 = GT\$$. Suffix set is calculated for each sequence. Map with each suffix and their occurrence count computed by *Algorithm* 1 is summarized in below Table. Combined length of all the sequences is $n = \sum_{i=1}^{n} |Si|$. For a given sequence *Si* number of suffixes is $|S_i|$ and can be computed in *O*|*Si*|. Hence suffixes of all sequences can be calculated in $O \sum_{i=1}^{n} |Si| = O(n)$.

In the second phase, we start with the empty tree and insert the suffixes from the map in sequence. If any suffix sequence is unique makes a completely new branch. If the partial matchingsequence is already found in the tree then till the match is found frequencies are summed up and for the unique portion, a new branch is formed *Algorithm II*. explains the process. Fig. 2(a) below shows result of inserting suffix *ACGT$* in empty tree. Next we insert a partial overlapping suffix *APGT$* Fig. 2(b).

Algorithm II: Generalized Suffix Tree Construction

Input: Map of all suffixes with frequency $m(suffix, count)$
Output: Generalized Suffix Tree
Algorithm:
For each $suffix_i, count_i \in m(suffix, count)$

I. Traverse tree till any branch has overlap with $suffix_i$ and increment each node by $count_i$
II. For remaining alphabets of $suffix_i$fork a branch starting last node till where overlap was found in step I.
For each node in new branch, set count equal to $count_i$.

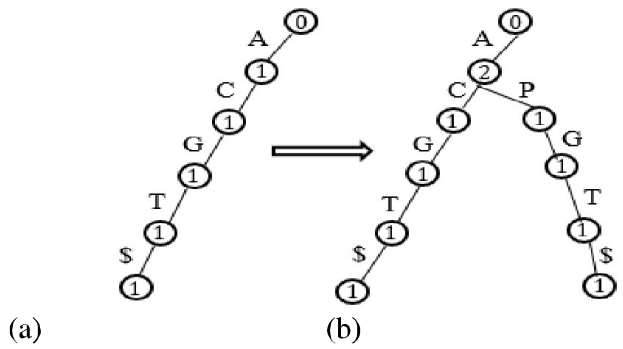

Figure 2: (a) String ACGT$ inserted in anempty tree. (b) Then string APGT$ inserted

At any point of time height of the tree is $h \leq \max(|S_i| \forall \; 1 \leq i \leq n)$ is the length of longest suffix. Total number of suffixes to be inserted in tree is $n = \sum_{i=1}^{n} |Si|$.. Each insertion starts with root and can go till leaf

i.e. total nodes to be traversed can be up to h. So complexity of this phase is $O(nh) \leq O(n^2)$ which dominates overall complexity.

V. DISTRIBUTED CONSTRUCTION OF PROBABILISTIC GENERALIZED SUFFIX TREE (PGST)

Map-reduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. Map-reduce works by breaking the processing into two phases: the map phase and thereduce phase. Each phase has key-value pairs as input and output [22]. Mapper procedure that performs filtering and sorting and a reducer procedure perform a summary operation.

Set of string sequences $S = S_1, S_2 \ldots . S_n$ is partitioned in proportion to number of computing nodes available in cluster. If number of nodes in clustered k then number of partitions created is $\frac{n}{k}$ where n is cardinality of S. For each of the partition of strings, a mapper module is triggered. Mapper locally operates on set a of sequeces assigned to it and output is the map of suffixes a keys and values as frequency of their occurrence similar to described *Algorithm I* in *section III*. For demonstration we take S contains following four strings $S_1 = ACGT\$, S_2 = GT\$, S_3 = ACGT\$, S_4 = ABGT\$$. If k=2 .then say two partitions got created as Partition I: $S_1 = ACGT\$, S_2 = GT\$$ and Partition II: $S_1 = ACGT\$, S_2 = ABGT\$$. Processing in mapper is described below. For each partition a mapper is invoked and two output from corresponding mappers are shown in below.

Mapper Module: Suffix frequency calculation of local partition

Input: partition of strings $S_1, S_2 \ldots . S_m$
Output: Map of all suffixes with frequency
$map_x(suffix, count)$
Algorithm:
1. For each string $S_i \in S_1, S_2 \ldots . S_m$ repeat I and II
   I. Compute all suffixes of S_i
   II. Insert suffixes calculated into map_x. If suffix exists in map_x increment count else insert suffix with $count$ **1** .
2.Send map_x to reducer

	$mapper_x$				$mapper_y$		
i	x_i	Suffix	Count	j	y_j	Suffix	Count
1	x_1	ACGT$	1	1	y_1	ACGT$	1
2	x_2	CGT$	1	2	y_2	CGT$	1
3	x_3	GT$	2	3	y_3	GT$	2
4	x_4	T$	2	4	y_4	T$	2
5	x_5	$	2	5	y_5	$	2
				6	y_6	APGT$	1
				7	y_7	PGT$	1

Table 4: All suffix and their frequency from mappers

Suffix tree build is taken care in reduce phase. The process starts from an empty tree. Each suffix from a different mapper is inserted in the tree. Since we add frequencies during each insertion overall frequency for each node remains same as when computed on a single node. For example, when computed on single node suffix has frequency 4 on each corresponding node of the tree. In reducer module for same suffix in mapper-I and mapper-II frequency is 2 in each. Reducer will insert this suffix twice and hence for this suffix frequency count will be 4. Performance on Hadoop cluster is shown in Figure 6.

Reducer Module: Generalized Suffix Tree Construction

Input: Maps of suffixes with frequency from mapper modules
$m_x(suffix, count) \ldots . . m_{x+k}(suffix, count)$
Output: Generalized Suffix Tree
Algorithm:
For each $suffix_i, count_i \in m_{x+j}(suffix, count) \forall 0 \leq j \leq k$
   I. Traverse tree till any branch has overlap with $suffix_i$ and increment each node by $count_i$
   II. For remaining alphabets of $suffix_i$ fork a branch starting last node till where overlap was found in step I. For each node in new branch set count equal to $count_i$.

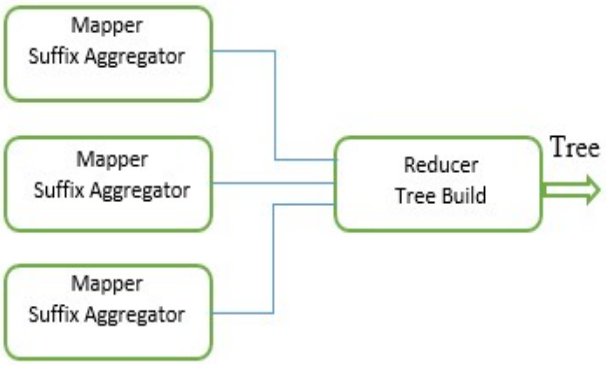

Figure 3: Summary of architecture of process

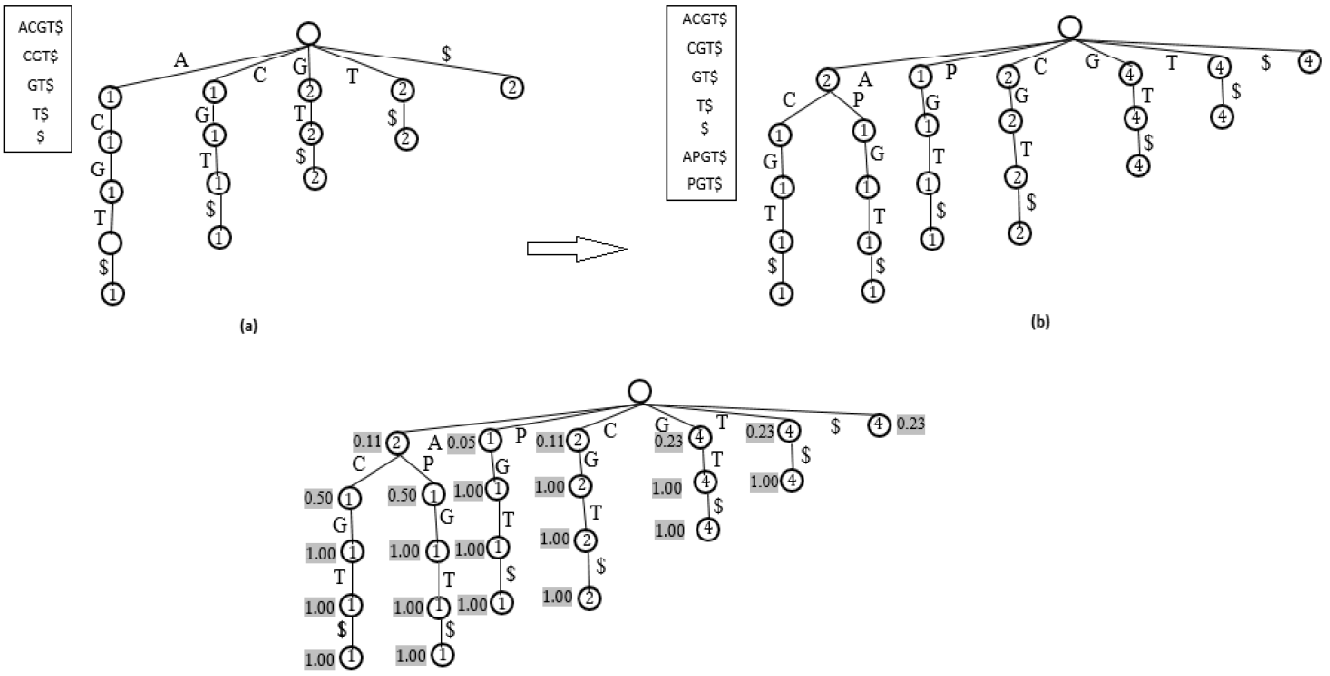

Figure 4: (a) Suffix tree after output suffixes from mapper$_x$ is inserted in empty tree along with frequency (b) Suffix tree after output suffixes from mapper$_y$ is inserted intotreeofprevious step & frequency merged (b) Suffix tree after probability computed

V. EVALUATION

This section presents performance evaluation results of distributed construction of probabilistic suffix tree construction on Reuters-21578 collection of news articles. It is most frequently used in theevaluationof NLP techniques especially classification. (http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html). It has around 12,902 documents with 90 classifications. We used these documents for PST evaluation. Each document is assumed to be an input stream of alphabets and PST is constructed. Data was stored on

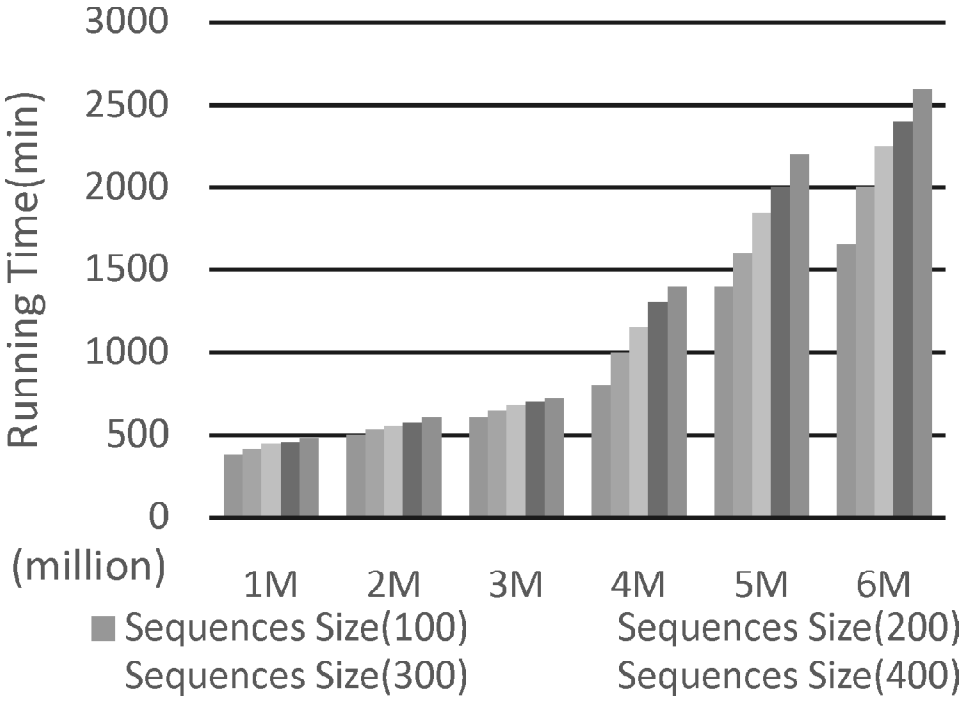

Figure 5: Performance with varying sequence size

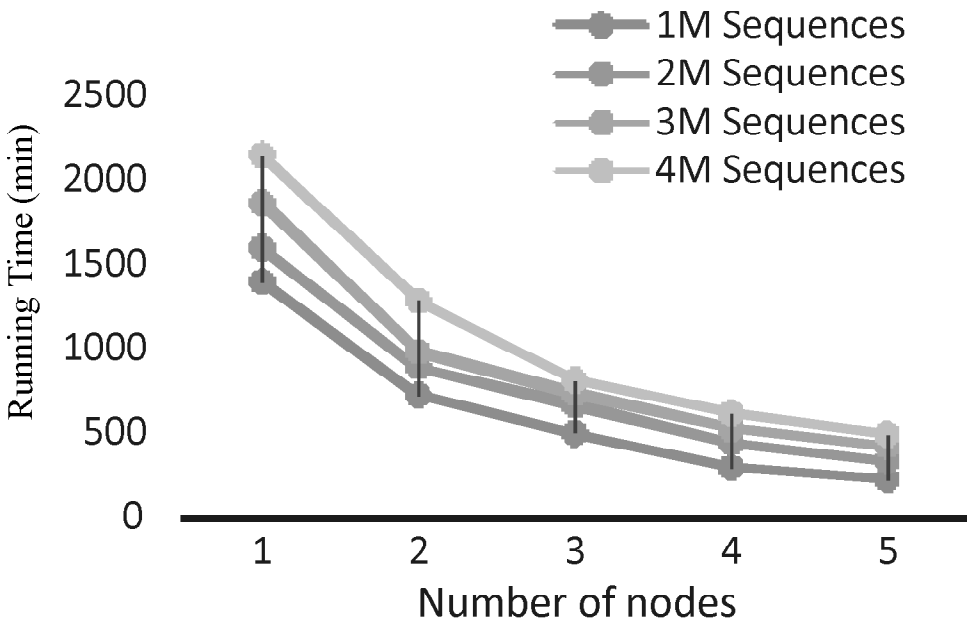

Figure 6: Performance on Hadoop cluster

Hadoop distributed file system (HDFS) distributed over thecluster. Replication factor affects time complexity. If it's too low then lots of time goes into network latency for data transfer to thecomputing node. To achieve better results we kept replication factor equal to number of compute nodes. So that data is available on all computation nodes and we don't spend too much time in data transfer. The experiment was performed over cluster containing 6 compute machines (five workers and one master node).Each node has a64-bit processor with 8 GB main memory. The result is proof of concept it scales linearly.

REFERENCES

[1] Begleiter, R., El-Yaniv, R., & Yona, G., "On prediction using variable order Markov models", Journal of Artificial Intelligence Research, 22, 385–421, 2004.

[2] M. Comin and M. Farreras, "Efficient parallel construction of suffix trees for genomes larger than main memory", EuroMPI'13, Proceedings of the 20th European MPI User's Group Meeting, 2013.

[3] Paul Bieganski; John Riedl; John Carlis; Ernest F. Retzel, "Generalized Suffix Trees for Biological Sequence Data". Biotechnology Computing, Proceedings of the Twenty-Seventh Hawaii International Conference on. pp. 35–44,1994.

[4] E. Mansour, A. Allam, S. Skiadopoulos, P. Kalnis, "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.

[5] U. C. Satish, P. Kondikoppa, S. Park, M. Patil, R. Shah, "Mapreduce based parallel suffix tree construction for human genome", 20th IEEE International Conference on Parallel and Distributed Systems ICPADS 2014 Hsinchu Taiwan, pp. 664-670, 201, 2014.

[6] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm", J. ACM, vol. 23, no. 2, pp. 262–272, 1976.

[7] E. Ukkonen, "On-line construction of suffix trees," Algorithmica, vol. 14, no. 3, pp. 249–260, 1995.

[8] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan, "On the sortingcomplexity of suffix tree construction," J. ACM, vol. 47, no. 6, pp. 987–1011, 2000.

[9] E. Hunt, M. P. Atkinson, and R. W. Irving, "A database index to large biological sequences," in In VLDB, 2001, pp. 139–148, 2001.

[10] S. J. Bedathur and J. R. Haritsa, "Engineering a fast online persistent suffix tree construction," in ICDE, pp. 720–731, 2004.

[11] C.-F. Cheung, J. X. Yu, and H. Lu, "Constructing suffix tree for gigabyte sequences with megabyte memory," IEEE Trans. Knowl. Data Eng., vol. 17, no. 1, pp. 90–105, 2005.

[12] B. Phoophakdee and M. J. Zaki, "Genome-scale disk-based suffix tree indexing," in Proceedings of the 2007 ACM SIGMOD international conference on Management of data, ser. SIGMOD '07. New York, NY, USA: ACM, pp. 833–844.

[13] P. Weiner, “Linear Pattern Matching Algorithms,” in SWAT, 1973, pp.1–11, 2007.

[14] R. Hariharan, “Optimal parallel suffix tree construction,” in STOC, pp. 290–299, 1994.

[15] G. M. Landau, B. Schieber, and U. Vishkin, “Parallel construction of a suffix tree (extended abstract),” in ICALP, pp. 314–325, 1987.

[16] S. Tata, R. A. Hankins, and J. M. Patel. “Practical suffix tree construction”, In Proc. of VLDB, pages 36–47, 2004.

[17] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel, ”Practical methods for constructing suffix trees”, The VLDB Journal, 14(3):281–299, 2005.

[18] A. Ghoting and K. Makarychev. “Indexing genomicsequences on the IBM blue gene. In Proceedings ofConference on High-Performance ComputingNetworking, Storage and Analysis (SC), pages 1–11,2009.

[19] M. Barsky, U. Stege, and A. Thomo. Suffix trees for inputs larger than main memory. Information Systems, 36(3):644 – 654, 2011.

[20] K. Erciyes, “Distributed and Sequential Algorithms for Bioinformatics”, Computational Biology, Springer International Publishing, ISBN 978-3-319-24966-7, 2015.