Improving the Run-Time of Space-Efficient n-Gram Data Structures Using Apache Spark

Fotios Kounelis, Andreas Kanavos, and Phivos Mylonas

Abstract

Storing information in memory efficiently is one of the most significant challenges in computer science. The two main factors that consist an efficient data structure is the reduction of space and time consumption. There is a plethora of different tools able to reduce the run-time of a process, and Apache Spark is one of these; it is a computing framework that is using clusters to execute a process. There are two key features in this software, a directed acyclic graph (DAG) that maps the execution process and the resilient distributed datasets (RDD), which allow large in-memory computations. In order to construct a data structure, which is space- and time-efficient, we have to utilize the corresponding framework. A comparison of the run-time improvement with the use of Spark is also provided. Finally, to prove the efficacy of this software tool, we construct a space-efficient data structure and compare the run-time with and without its use.

Keywords

Inverted files · n-gram indexing · Computing performance · Apache Spark · Biological sequences

1 Introduction

Modern applications demand the efficient processing of information, and in com-bination with ever increasing data, scientists should find more efficient software that will reduce the time and space consumption. Two major problems arise for computer scientists, namely, how to store all this information and the way to process it. Major research works have been published in both fields to find algorithms that can solve these problems. In the area of space compaction, a large amount of data structures has been proposed like su x trees [17]. This data structure is a tree structure that can store data by storing all the su xes of the words [15]. Also, inverted index is a data structure that is able to store big texts with the use of a lexicon and inverted lists [1, 3]. There are also

F. Kounelis (✉)
Department of Computing, Imperial College London, London, UK
e-mail: f.kounelis20@imperial.ac.uk

A. Kanavos (✉)
Computer Engineering and Informatics Department, University of Patras, Patras, Greece
e-mail: kanavos@ceid.upatras.gr

P. Mylonas (✉)
Department of Informatics, Ionian University, Corfu, Greece
e-mail: fmylonas@ionio.gr

P. Vlamos (ed.), *GeNeDis 2020*, Advances in Experimental Medicine and Biology 1338,
https://doi.org/10.1007/978-3-030-78775-2_19

many improvements which aim at reducing the space demand for storing big data.

On the other hand, many different software tools implementing different algorithms have been developed in the last years to reduce the time consumption of a process. One of these tools is Apache Spark [18], which uses a direct acyclic graph and resilient distributed datasets. This open-source software breaks the process into different parts, operates every different part at the same time, and concludes to the outcome. Except for Apache Spark, there are many different tools, like Apache Hadoop, which use different techniques and rapidly analyze the data.

The origin of the data, in our days, can be from almost everywhere. The rapid increase in technology use leads to a large amount of data in any field. In this paper, we are interested in biological data where, following [9] that construct a space-efficient data structure to store biological sequences, different lengths of biological sequences to provide the run-time improvement are processed.

Nowadays, there is an increasing interest in understanding the human genome as well as the way living organisms are functioning. As a result, different genomics faculties have been created to study this field; these faculties export a large amount of data that can be DNA sequences, protein sequences, etc. The aim is to export the information from the organisms and then process and analyze them. In this paper, we want to provide the most efficient way for these faculties to work with and reduce their processing time by using the best tools.

The data structure that is constructed in this paper to compare the run-time is a combination of su x trees along with inverted indexes. Su x trees provide much information for a stored string, whereas the feature that we take advantage of is the number of occurrences of a subsequence inside the string. After that, we use inverted indexes and store each subsequence depending on the number of occurrences that were produced using the previous structure.

The remainder of the paper is structured as follows: Sect. 2 overviews work that has been published on run-time efficiency and Apache Spark. Section 3 presents in detail the data structure and the way it is constructed, while in Sect. 4, implementation details along with dataset used and Spark configuration are introduced. The experiments conducted with the corresponding discussion are described in Sect. 5. Ultimately, Sect. 6 presents conclusions and directions for future work that may extend the current version and performance of this work.

2 Related Work

In [2], a set of efficient algorithms for string problems arising in the computational biology area were presented, adapting traditional pattern matching techniques to the weighted scenario. There is a connection with the probabilistic su x tree, where every node is associated with a probability vector that stores the probability distribution for the next symbol, given the label of the node [14].

In addition, our representation of n-grams and our space compaction heuristics are of general nature concerning the efficient handling of multilingual documents in web search engines and information retrieval applications. In [11], regarding some languages, the characters are more like syllables than letters, and most words are small in numbers of characters; so, it is better to use n-grams.

The rapid development of next-generation sequencing (NGS) technology has generated a large amount of sequence data, which has a tremendous impact on sequence alignment and mapping processes. Specifically, for a detailed survey regarding Spark-based applications used in NGS and other biological domains, the work in [5] can be taken into consideration. Further applications include assembly, sequence analysis, phylogeny, and single-cell RNA sequencing.

StreamBWA is a streaming distributed strategy where the input files were streamed into the Spark cluster [12]. This reduces the time required to preprocess data and combine the nal results. Meanwhile, authors in [10] introduced GATK-Spark, a balanced parallelization approach that implements an in-memory version of GATK using Apache Spark. This paralleled the GATK

pipeline by taking full account of computation and workload. Another similar approach is presented in [6], where authors tried to migrate existing programs to multinode clusters without changing the original programs using Spark. The intermediate data are immediately consumed again on the nodes that generated the data, reducing time and network bandwidth consumption. In addition, a new assembling algorithm based on Spark, which aimed at simplifying the De Bruijn graph, was proposed in [13].

This paper builds up on the works presented in [4, 16], where a set of algorithmic techniques for efficiently handling weighted sequences by using inverted files were discussed. Specifically, these methods deal effectively with weighted sequences using the n-gram machinery and act as alternatives to other techniques that mainly use su x trees. Furthermore, authors introduced a general framework that can be employed so as to reduce the space complexity of the two-level inverted files for n-grams.

3 Proposed Method

Initially, in our methodology, the data structure introduced in [9] is utilized. A biological sequence is inserted into a su x tree, which will store all the su xes of this sequence. It provides the number of occurrences for each selected subsequence of a certain length. Based on each subsequence calculated occurrences, Eq. 1 stores the subsequences on two different inverted indexes. This equation calculates the average amount of occurrences, namely, T, of a subsequence with length len in a string with N characters, where k is the size of the alphabet:

$$T = \text{len}^2 \frac{N - \text{len} + 1}{k^{\text{len}}} \tag{1}$$

In the following, we calculate the weight of each subsequence based on the product of the subsequence length and the number of times it occurs in the string. The subsequences that have a fewer number of occurrences than the threshold of Eq. 1 are stored in a classic inverted index. The subsequences that have a higher number of occurrences are stored in a two-level inverted index [8].

The classic inverted index is a structure that consists of two parts. Firstly, there is a dictionary, also called lexicon, which stores all the words of a text in alphabetical order. Each word of the dictionary is then connected to a list, which consists the second part of the structure. This posting list contains the positions inside the text that this word occurs. By using this structure, we can time-efficiently reply to position queries for a certain word by reading the connected list for this word.

The two-level inverted index consists of dictionary and posting lists but has a slightly different structure as it stores the data in two phases. In the first phase, it builds an inverted index, also called back-end index, which stores the different subsequence and their occurrences. Following our previous work [9], we have utilized the same number of subsequences regarding the back-end index. Hence, it is built for seven different lengths of subsequences, from 4 to 10.

During the second phase of the algorithm, a second inverted index is constructed, also called the front-end index. The lexicon of the front-end index consists of all the n-grams extracted from the subsequences of the back-end index. The front-end index also consists of the inverted lists, which have the information of the subsequence that the relevant n-gram occurred. We construct three different front-end indexes for each back-end index – a bigram, a trigram, and a four-gram for three different variations of n, which are two, three, and four, respectively.

3.1 Two-Level Inverted Index Example

To better understand the two-level inverted index, we present the following example in which there is a DNA sequence, with the letters A, G, C, and T. Let us examine the sequence TGATGCAGGTCTG, with length N = 13, and hence, all the subsequences with len = 4 will be produced. We have also elongated the subsequences by n 1 so as not to lose any n-gram.

Table 1 introduces the two-level inverted index, whereas Table 1a presents the back-end index created for this sequence. There are four different subsequences extracted for the examined length and elongation. The dictionary, which is con-structed for the back-end index, is in alphabetical order and for each subsequence, a posting list, which specifies the position of the word inside the sequence, exists. In particular, the first entry of the dictionary specifies that the subsequence "AGGT" occurs in position 6 of the first sequence (since we have only one sequence in our example, all subsequences have the same identifier 0, and the first character is at position 0).

In the next phase, the front-end index is constructed based on the back-end index. More specifically, Table 1b displays all different n-grams extracted from Table 1a. For this example, we extract only the n-grams having a length equal to 2, namely, bigrams. Each extracted n-gram exists in the front-end index in alphabetical order. For each n-gram, a posting list that specifies the subsequence of the back-end index, and the position inside each subsequence is considered. In particular, the last entry of Table 1a, i.e., the bigram "TG", occurs in three different subsequences of the back-end index, namely, in position 2 of the subsequence 1, in position 0 of the subsequence 2, and in position 0 of the subsequence 3 (the starting position of the subsequences of the back-end index starts at 0).

4 Implementation

We implemented our methods using the Python language, and the time consumption in each case is counted. In particular, we measure separately the construction of each one of the seven back-end indexes as well as the generation and separation between the classic and two-level inverted index for each subsequence. In addition to that, we also calculated the time for each separate front-end index that was developed. As a result, for each process, we keep track of 35 different run-times, and we compare them with all the other multithreading processes.

4.1 Dataset

The above procedure was measured and compared to the three different node levels. Apart from that, we study different file sizes as we take biological sequences from living organisms[1] with sizes equal to 5 MB and 20 MB. Firstly, with the use of different sizes, we aim to study the fluctuation of the run-time improvement for each one separately. In the following, we check if there is any difference in the improvement of different sizes to propose the best file sizes for such methods.

4.2 Apache Spark Framework

Apache Spark framework[2] [7] is a newer framework built in the same principles as Hadoop. While Hadoop is ideal for large batch processes, it drops in performance in certain scenarios, as in iterative or graph-based algorithms. Spark, in contrast to Hadoop, maintains the data in the workers' memory, and as a result it outperforms the later in algorithms that require many operations.

Our cluster includes 16 computing nodes (VMs), each one of which has 4 2:4 GHz CPU processors, 16 GB of memory, and 120 GB hard disk. One of the VMs serves as the master node and the other 15 VMs as the slave nodes.

Moreover, we apply the following changes to the default Spark configurations: 12 total executor cores (4 for each slave machine) are used; the executor memory is set equal to 8 GB and the driver memory to 4 GB. The above configurations were applied in the case of 16-node experiments. Regarding the four-node experiments, one of the VMs serves as the master node and the other three VMs as the slave nodes.

[1] ftp://ftp.ncbi.nih.gov/genomes/
[2] http://spark.apache.org/

Table 1 Two-level inverted index

(a) Back-end index

AGGT	0,[6]
TCTG	0,[9]
TGAT	0,[0]
TGCA	0,[3]

(b) Front-end index

AG	0,[0]		
AT	2,[2]		
CA	3,[2]		
CT	1,[1]		
GA	2,[1]		
GC	3,[1]		
GG	0,[1]		
GT	0,[2]		
TC	1,[0]		
TG	1,[2]	2,[0]	3,[0]

5 Results

In terms of a computer system, two major issues are program run-time along with space consumption. So, the main goal of our paper is to reduce the time needed of a two-level inverted index, based on n-grams for a biological sequence [8]. In [9], a data structure was implemented to improve the space demand, and with the use of multithreading, this high space efficiency is kept by achieving a less time-consuming process.

Our approach is based on the comparison in terms of a cluster with a variety of nodes. Specifically, the experimental results of the classical approach using 1 node are compared with the results of a cluster with 4 and 16 nodes. As expected, as the number of nodes is increasing and the process is diverted in more threads, the run-time will be reduced, although the factor of improvement is not linear for all thresholds and input files. In the following, the results for 5 MB and 20 MB data files for seven different lengths of subsequences are discussed.

5.1 Dataset of 5 MB

Table 2 displays the time, for the different lengths of subsequences, the threshold equation, the back-end index, and the three different front-end indexes for the input file of 5 MB.

We observe that the execution time of the weight calculation and the separation of the subsequences, namely, threshold equation, for length equal to 4 is reduced from 9:28 s for 1 node to 5:15 s for 4 nodes and 3:22 s for 16 nodes. This leads to a 44% decrease between 1 and 4 nodes and a 37% decrease between 4 and 16 nodes. While the nodes are increasing by a factor equal to 4 in each node set, we observe that the percentage fluctuates and is not linear, i.e., 44% and 37%. Moreover, this difference between the percentage decrease lies also between the different length of subsequences for the same node sets, i.e., in the threshold equation phase, for length 4, the percentage decrease between one and four nodes is 44%, while for length 10, we have a decrease of 29% (20,821:17 s and 14,794:61 s for 1 and 4 nodes, respectively).

Similar to the threshold equation phase, the execution time to construct the back-end index reduces as the number of nodes increases. Based on that, for each node set, the higher the length of the subsequence, the more the execution time. More specifically, for one node and length of subsequence equal to 6, the execution time is 40; 81 s, which is lower than length 10, i.e., 7421; 36 s. The corresponding values for 4 nodes are 28:45 s and 6292:14 s, while for 16 nodes 19:82 s and 4198:78 s. This substantiates our expectation for lower execution time as the number of nodes increase. However, the percentage of the decrease between each node set fluctuates more than one

Table 2 Execution time for 5 MB (in seconds)

Length of substring	Threshold equation	Back-end	Front-end 2-gram	Front-end 3-gram	Front-end 4-gram
		1 node			
4	9.28101	7.6873	0.03127	0.01562	0.01562
5	22.87442	14.9684	0.03121	0.01564	0.03127
6	79.88853	40.81274	0.09374	0.0625	0.0625
7	309.02449	135.29478	0.31246	0.18749	0.14065
8	1245.51827	500.14972	1.14055	0.79689	0.82807
9	5073.25390	1873.86071	4.7499	3.43738	2.74993
10	20821.17229	7421.36633	19.03075	16.10894	11.34345
		4 nodes			
4	5.15471	5.08626	0.07064	0.04040	0.03893
5	16.25639	10.44123	0.17678	0.10057	0.0418
6	56.31926	28.45171	0.12700	0.06289	0.05842
7	215.99530	94.85067	0.22914	0.12222	0.10695
8	872.44428	357.68164	0.77126	0.44001	0.45133
9	3564.83482	1394.52719	2.77309	1.85139	1.48160
10	14794.6182	6292.14491	11.86227	9.13899	6.34939
		16 nodes			
4	3.224	2.86304	0.062154	0.03541	0.05124
5	11.93713	6.90567	0.10548	0.085694	0.09548
6	30.91816	19.82573	0.01025	0.03986	0.04553
7	156.03783	62.21554	0.18422	0.07931	0.076
8	573.19352	234.42779	0.47523	0.25477	0.3172
9	2661.85641	978.50024	1.95494	1.19603	1.09481
10	10727.26266	4198.78216	8.39711	5.08691	3.6512

would expect. In particular, there is a 33% decrease for length 4 between one and four nodes (7:68 s and 5:08 s, respectively), while for the same nodes, the percentage for length 10 is 15%.

In contrast to the robust behavior of the threshold equation and back-end index phase, the front-end construction phase execution times vary. For each node set, the execution time increases as the length of the subsequence increases, although this does not stand for the increase of the number of nodes. In particular, the execution time using one node, for length of subsequence equal to 5 and the front-end consists of 4 grams, is 0:03 s. For the same features and length equal to 10, the execution time increases to 11:34 s. When the number of nodes increases to four, the respective times are 0:04 s and 6:34 s. This shows that regarding length 10, the execution time reduces, whereas regarding length 5, the execution time increases. This can be explained by the fact that the run time for such small subsequences is really small and the overhead cost varies leading to such results. The same fluctuated behavior is observed for the 16 nodes set where the time for the length of subsequence 5 is 0:09 s and for 10 is 3:65 s.

The same behavior is noticed for all front-end indexes. When the execution time is very low, the results may not be as robust as expected. Our final observation is that as the length of the n-gram increases, the execution time reduces. This is an expected behavior based on the construction phase of the front-end index; to construct the front-end index, each of its subsequence is split into the desired n-grams, and as a result, the higher the n-gram, the fewer the splits on the subsequence.

Table 3 Execution time for 20MB (in seconds)

Length of substring	Threshold equation	Back-end	Front-end 2-gram	Front-end 3-gram	Front-end 4-gram
		1 node			
4	34,43379	30,05907	0,01501	0,01003	0,01197
5	93,82897	63,54991	0,04097	0,02197	0,02097
6	326,63004	173,70683	0,08901	0,07302	0,05900
7	1271,08973	579,21976	0,29898	0,20802	0,16402
8	5078,52252	2160,27892	1,33792	0,93194	0,86494
9	20706,64239	8074,79021	4,91471	3,65278	3,15581
10	84400,59777	30325,55520	19,6448	15,94504	11,08534
		4 nodes			
4	21,90268	20,48459	0,17766	0,06445	0,05416
5	63,09541	45,94116	0,09985	0,04239	0,04498
6	229,08398	121,66234	0,26465	0,07040	0,07472
7	892,55925	409,91803	0,38385	0,16764	0,14046
8	3607,09192	1534,54903	0,96597	0,62060	0,60317
9	14753,43285	5745,00896	3,76170	2,56384	2,31665
10	60390,94628	21626,81785	12,69483	9,39571	6,72134
		16 nodes			
4	14,442	14,5593	0,10543	0,076548	0,085647
5	41,3526	31,91015	0,15236	0,06235	0,05487
6	171,21432	79,22427	0,23300	0,04958	0,75054
7	575,55304	282,86946	0,30546	0,10325	0,10206
8	2375,15663	1022,53792	0,68256	0,42454	0,43478
9	10976,8498	4252,21882	2,68825	1,76771	1,76723
10	41854,88216	15067,96253	8,22551	5,90836	4,04803

5.2 Dataset of 20 MB

Table 3 displays the time for the different lengths of subsequences, the threshold equation, the back-end index, and the three different front-end indexes for the input file of 20 MB.

Initially, we notice that the behavior for the threshold equation and back-end index is robust and similar to the 5 MB file. For example, the execution time in the threshold equation phase for length of subsequence equal to 4 and one node is 9:28 s, while for length 10 is 20,821:17 s. In addition, we can identify a steep increase in the execution time as the length increases. In addition to that, for four nodes and length of subsequence equal to 4 and 10, the execution time is 5:15 s and 14,974:61 s, respectively. This steep increase of the execution time between the different lengths can also be seen in terms of one node. However, the execution time is lower than corresponding lengths, which describes the time reduction when the number of nodes increases. Once the nodes are increased to 16, the execution time further reduces to 3:22 s and 10,727:26 s for length values equal to 4 and 10, respectively.

The main difference for threshold equation and back-end index phases between the 5 MB and 20 MB file is that the latter has a smaller fluctuation of the reduction percentage as the nodes increase. More specifically, in the 5 MB file, we observe that the highest percentage is 44% for the configuration of threshold equation phase, for length of subsequence equal to 4 and one to four nodes, while the lowest is 15% for the configuration of back-end phase, for length of subsequence equal to 10 and 1–4 nodes. In contrast, for the 20 MB file, the highest percentage is 36% for the configuration of threshold equation phase, for length of subsequence equal to 4 and 1–4 nodes, while the lowest is 25% for the configuration of threshold equation phase, for length of subsequence equal to 6 and 4–16 nodes. Furthermore,

there is a more robust behavior when the node increases from one to four, which means that the execution time reduction will be almost the same for the lengths 4 and 10. Hence, we conclude that the higher the data file is, the more robust the fluctuation for the different lengths of subsequence will be.

The front-end construction phase for the 20 MB file follows the same behavior as the 5 MB file. As the length of the subsequence increases for a node set, so does the execution time, although the execution time does not reduce as the number of nodes increases, as in the back-end or the threshold equation phase. This occurs when the execution time is very low, under 1 s, and can be explained by the overhead cost that is added to these values.

A final observation on the execution time between the two different tables can be drawn. As the amount of data increases from 5 MB to 20 MB, the execution time for phases, like the threshold equation or back-end index, is increased for at least four times. However, the execution time for the construction of the front-end index remains almost the same, with a difference of up to 1 s. This is a very interesting result, as it proves that the front-end index construction will remain almost the same for different data files.

6 Conclusions and Future Work

In the proposed paper, we compare the run-time of algorithmic techniques set for efficiently handling biological sequences by using inverted files. Following previous works, we implemented a data structure to improve space demand with the use of multithreading. Apache Spark framework was utilized in order to prove the efficacy of the proposed schema, and the results of this space-efficient data structure are quite promising. These methods deal effectively with DNA sequences using the n-gram machinery and act as alternatives to other techniques that mainly use su x trees.

Regarding future work, the proposed methodology can be augmented with additional datasets to establish a better understanding. In addition, the cluster on which we tested our experiments is another key aspect of cloud computing, in general, to better evaluate Spark's performance in terms of time and scalability. Finally, a limitation that illustrates the restrictions of our proposed methodology and suggests room for improvement in future studies is the inverted file intersection algorithms utilization along with the incorporation of some extra data structures to test the time efficiency of our scheme when handling such queries.

References

1. Baeza-Yates RA, Ribeiro-Neto BA (2011) Modern information retrieval: the concepts and technology behind search, 2nd edn. Pearson Education Ltd., Harlow
2. Christodoulakis M, Iliopoulos CS, Mouchard L, Perdikuri K, Tsakalidis AK, Tsichlas K (2006) Computation of repetitions and regularities of biologically weighted sequences. J Comput Biol 13(6):1214–1231
3. Cutting DR, Pedersen JO (1990) Optimizations for dynamic inverted index maintenance. In: 13th International Conference on Research and Development in Information Retrieval (SIGIR), pp 405–411
4. Diamanti K, Kanavos A, Makris C, Tokis T (2014) Handling weighted sequences employing inverted files and su x trees. In: 10th International Conference on Web Information Systems and Technologies (WEBIST), pp 231–238
5. Guo R, Zhao Y, Zou Q, Fang X, Peng S (2018) Bioinformatics applications on apache spark. GigaScience 7(8)
6. Harnie D, Saey M, Vapirev AE, Wegner JK, Gedich A, Steijaert MN, Ceulemans H, Wuyts R, Meuter WD (2017) Scaling machine learning for target prediction in drug discovery using apache spark. Futur Gener Comput Syst 67:409–417
7. Karau H, Konwinski A, Wendell P, Zaharia M (2015) Learning spark: lightning-fast big data analysis. O'Reilly Media
8. Kim M, Whang K, Lee J, Lee M (2005) n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In: 31st International Conference on Very Large Data Bases (VLDB), pp 325–336
9. Kounelis F, Makris C (2017) Space efficient data structures for n-gram retrieval. AIMS Med Sci 4(4):426–440
10. Li X, Tan G, Zhang C, Li X, Zhang Z, Sun N (2016) Accelerating large-scale genomic analysis with spark. In: IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp 747–751

11. Manning CD, Raghavan P, Schutze H (2008) Introduction to information retrieval. Cambridge University Press
12. Mushtaq H, Ahmed N, Al-Ars Z (2017) Streaming distributed DNA sequence alignment using apache spark. In: 17th IEEE International Conference on Bioinformatics and Bioengineering (BIBE), pp 188–193
13. Pan X, Fu XL, Dong GF, Li HH (2016) DNA sequence splicing algorithm based on spark. In: International Conference on Industrial Informatics - Computing Technology, Intelligent Technology, Industrial Information Integration (ICI-ICII), pp 52–56
14. Sun Z, Yang J, Deogun JS (2004) MISAE: A new approach for regulatory motif extraction. In: 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB), pp 173–181
15. Ukkonen E (1995) On-line construction of su x trees. Algorithmica 14(3):249–260
16. Volis G, Makris C, Kanavos A (2016) Two novel techniques for space com-paction on biological sequences. In: 12th International Conference on Web Information Systems and Technologies (WEBIST), pp 105–112
17. Weiner P (1973) Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory (SWAT), pp 1–11
18. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016) Apache spark: a unified engine for big data processing. Commun ACM 59(11):56–65