DLA: a Distributed, Location-based and Apriori-based Algorithm for Biological Sequence Pattern Mining

Eirini Stamoulakatou *
Politecnico di Milano, Italy
eirini.stamoulakatou@polimi.it

Andrea Gulino *
Politecnico di Milano, Italy
andrea.gulino@polimi.it

Pietro Pinoli
Politecnico di Milano, Italy
pietro.pinoli@polimi.it

***Abstract*—With the rapid growth of genomic data, the need for scalable data mining algorithms has increased. Frequent contiguous sequence mining is a technique that can help biologists to better understand the function and structure of our DNA, by capturing the common characteristics among related sequences. Many sequence mining algorithms have been developed over time. However, most of them suffer from scaling issues when dealing with big data or give no warranty for the completeness of their result. In this paper, we propose a distributed sequential pattern mining algorithm implemented on Apache Spark. Specifically, the algorithm exploits the Apriori Property and information about each patterns location within the original sequence, to drastically reduce the number of candidates at each iteration. Experimental results on real-world datasets confirm our performance expectations, showing a better scalability when compared to other distributed solutions.**

***Keywords*-Big Data; data mining; bioinformatics; high performance computing; sequential pattern mining; MapReduce;**

I. Introduction

Nowadays, sequence pattern mining (SPM), which finds frequent subsequences as patterns in a database of sequences, has become a fundamental data mining technique for extracting interesting patterns from a variety of data in different domains. Popular applications of SPM aim to predict, for instance, the customer behaviour, through the analysis of the purchased items[1] or the clickstream produced by customers while surfing a website. Other challenging applications can be found in bioinformatics, where huge amounts of biological sequences (e.g., DNA sequences[2]) can be processed to identify, for instance, the genetic components of a variety of diseases (e.g. cancer).

In bioinformatics, terabytes of DNA sequences can now be generated within a few hours with the use of next generation sequencing (NGS) technologies such as Illumina HiSeq X and Illumina Genome Analyzer. Frequently recurring patterns, called motifs, are found inside the DNA sequences. These sequences may be present because they have been conserved due to some mediating important biological functions. Sequence motifs are becoming increasingly important in the analysis of gene regulation, classification (using the frequencies as features for unsupervised learning methods) and also in drug design. Even thought several efficient SPM algorithms have been developed during the years, most of them are dated and lack of parallel implementations that suit the needs of the modern "big data" phenomenon.

In this paper we present DLA[3], a parallel and scalable algorithm for frequent continuous pattern mining, implemented on top of Apache Spark[4]. DLA mixes the philosophy and scalability of the popular WordCount problem's solution with the efficiency brought by location-based techniques for mining large sequence databases. Our location-based approach keeps track of the positions of each candidate frequent pattern within the original database of sequences. The location information associated to frequent patterns of length n is then used to efficiently generate candidate patterns of length $n + 1$. Since the expression "sequential pattern mining" addresses several computational problems, we provide an informal definition of the SPM problem solved by our algorithm through the following statement: given a sequence database (e.g. a database of strings) we aim to discover frequent contiguous sub-sequences (e.g. substrings), satisfying a predefined threshold.

In bioinformatics, sequences are usually defined over small alphabets. Hence, the identification of patterns from long genomic sequences requires flexible constraints, such as the definition of a minimum and maximum length for the pattern or a lower bound for the number of occurrences of a pattern within the database.

The rest of this paper is organized as follows: in Section I.A we present and overview of the existing SPM algorithms, then in Section II we provide a formal definition of the problem and the Apriori property while in Section III we describe the algorithm in detail, providing the example of execution on a small dataset. In Section IV we evaluate the performance of the algorithm and compare it with the performance of another parallel algorithm based on a different approach. Section V draws conclusions and closes the manuscript.

*These authors contributed equally to this work.

[1]Market Basket Analysis

[2]DNA stands for deoxyribonucleic acid and it is made of a very long sequence of 4 nucleotides that can be represented with the characters A (Adenine), C (Cythosine), G (Guanine) and T (Thymine)

[3]The acronym comes from Distributed, Location-based, Apriori-based

[4]https://spark.apache.org/

978-1-5386-5035-6/18/$31.00 ©2018 IEEE

A. Literature review

In this Section, we review the sequential pattern mining techniques designed for running on a single machine as well as distribution-based and cloud-based methods. It is possible to classify the traditional sequence mining methods into Apriori-based and pattern growth-based algorithms.

AprioriAll [1] is a level wise search algorithm based on the idea that as soon as an itemset is known to be infrequent, none of its super-sets has to be considered anymore [2]. Well known Apriori based algorithms are: GSP [3], SPADE [4], SPAM [5]. All the above algorithms are for small datasets that can fit in memory on a single machine and may not be suitable for big data mining in which datasets are too big to fit in memory or in a single node. FreeSpan [6] and PrefixSpan [7] are pattern growth-based algorithms. The general idea of PrefixSpan (Prefix-projected Sequential pattern mining), is to examine only the prefix subsequences and project only their corresponding postfix subsequences into projected databases. In each projected database, contiguous sequences are grown by exploring local length-1 frequent sequences. However, when mining long frequent concatenated sequences, this method is inefficient.

Moreover, numerous distributed algorithms have been proposed based on Apriori and pattern growth models [8], [9], [10]. As a gap-constraint frequent sequence mining algorithm implemented on MapReduce, MG-FSM [11] partitions the input database in a way that allows each partition to be mined independently using any existing frequent sequence mining algorithm. Morever the notion of w-equivalency w.r.t. a "projected database", which is used by many SPM algorithms, is introduced. In Section IV, the performances of MG-FSM and DLA are compared.

II. Problem Statement

We present the basic notions needed to clearly define the problem of sequential pattern mining.

Consider an alphabet Σ, i.e., a finite set of symbols; a sequence s over Σ is defined as a finite enumerated collection of elements of Σ:

$$s =< s1, s2, \ldots, s_n >, \qquad s_i \in \Sigma.$$

The length of the sequence s, denoted as $|s|$, is defined as the number of elements in the sequence.

Given two sequences $s =< s1, s2, \ldots, s_n >$ and $t =< s1, s2, \ldots, s_m >$, such that $|s| = n$, $|t| = m$ and $|m| < |n|$, we say that t is a sub-sequence of s or that s contains t, and we denote it as $t \subset s$, if there exists an integer i such that:

$$s_{i+j} = t_j \text{ for } j = 1, 2, \ldots, m.$$

A *sequence database* D is defined as a finite set of pairs $< id, s >$, also referred to as items, where each s is a sequence of arbitrary length on the same alphabet Σ and each id is a unique identifier of the corresponding sequence. Notice that a sequence database may contain two items with the exact same sequence.

Definition 1 (support of a sequence): Consider a sequence database D and a sequence s on the same alphabet. The support of the sequence s on the database D is the number defined by the function:

$$\text{sup}(s, D) = |\{< id, x >\in D : s \subset x\}|.$$

Thus, the support of a sequence over a database is the number of items in the database that contain the sequence.

For a database D and a minimum support value σ, the *pattern mining* is the problem of identifying all the sequences which support on D is at least σ. Usually, the problem is further constrained allowing only for sequences with at least a minimum length μ, since short sequences may appear more frequently. Sometimes, also a limitation on the maximum sequence length is set; we denote this parameter with α.

Definition 2 (Apriori property): Let D be a sequence database on an alphabet Σ and s and t any two sequences on the same alphabet. Then,

$$t \subset s \implies \text{sup}(t) \geq \text{sup}(s).$$

In other words, if a pattern s is frequent and t is a subsequence of s, then t has to be at least as frequent as s. Conversely, if a sequence t is infrequent, all of the sequences s such that s contains t must be infrequent.

In the algorithm we propose, we leverage this implication when we build new candidate sequences: a potentially frequent sequence can only be made by concatenating (or intersecting) shorter frequent sequences. This property is of primary importance, since it helps to considerably reduce the search space.

III. Algorithm description

Here we describe DLA, the algorithm we propose for sequential pattern mining. It is based on the Apriori property and can be summarized in three iterative steps. Each iteration of the algorithm looks for patterns of a specific length, namely n, starting from the user-provided minimum length μ. The algorithm proceeds as follows:

1) An initial set of *candidate patterns* is generated as the set of all the sub-sequences of length μ that can be found at least once in the database;
2) iteratively:
   a) supports of candidate patterns are computed;
   b) candidates whose support does not satisfy the threshold are discarded;
   c) retained candidates are used to generate candidates of length $n + 1$ and become the input of the next iteration;
3) the algorithm terminates when at least one of the following conditions is met:

- the set of candidates satisfying the support and frequency thresholds is empty; or
- $n > \alpha$, i.e. when the considered length exceeds the user-provided maximum length.

In the above algorithm, the most critical aspect for the performance is the generation of the next-length set of candidates; the amount of data to be processed by the next iteration depends on the adopted candidate generation strategy. At this stage, we can exploit the Apriori property which states that valid candidates for the next iteration can only be generated as a concatenation of the candidates of the current iteration. The correctness of the proposed strategy comes from the Apriori property. We present three main strategies for performing these steps of the algorithm. In order to easily understand their functioning, we consider the following sequence database:

seq-id	**seq**
1	ATCT
2	GT

assuming $\mu = \sigma = 1$ and that we already found the set of candidates of length 2 satisfying the thresholds,

$$c_2 = \{AT, TC, CT, GT\},$$

the possible strategies for candidates generation are then the following:

- *Blind*: all the sub-sequences of length 2 are joined, i.e. c_3^B is obtained performing the following joins:

Joined	Candidate
ATxTC	ATC
TCxCT	TCT
CTxTC	CTC
GTxTC	GTC

- *Sequence-based*: only couples of candidates produced by the same sequence are joined. In this case, sequence 2 will not contribute to the generation of any new candidate and the new set of candidates c_3 is:

  $$c_3^S = \{ATC, TCT, CTC\}$$

- *Location-based*: only couples produced by the same sequence and so that the position of the first candidate immediately follows the position of the second candidate are joined. In this case, candidate TC will not be joined with CT, since the former comes after the latter. Therefore,

  $$c_3^L = \{ATC, TCT\}$$

  Note that, with this approach, comparison between strings is not necessary.

Since the location-based strategy produces only candidates actually appearing in the sequence database, it is the strategy that produces the minimal number of new candidates; in particular the following property holds:

$$|c_3^L| \leq |c_3^S| \leq |c_3^B|$$

DLA is an Apriori based and uses a location-based candidate generation strategy.

A. Parallel implementation of the algorithm

The parallel implementation of the algorithm relies on the Apache Spark framework. Apache Spark is a general-purpose data processing engine providing high-level data operators and making a more efficient use of memory as compared with low-level map-reduce programming. The programming model of Spark is based on an abstraction called Resilient Distributed Dataset (RDD); a RDD is a distributed, fault tolerant data collection which can be processed on large servers or clusters. In Spark, all jobs are expressed as: creation of a new RDD, transformation of an existing one, or combining two RDDs in a new one. The three main operators used by our implementation are:

- *flatMap*: transforms an RDD producing, for each of its entries, 0 or more entries in the output RDD. This transformation is applied in parallel at each partition.
- *reduceByKey*: applies an associative binary function to the elements of the input RDD having the same key. The operation is first applied in parallel at each partition, and then applied to the aggregates with the same key in a single partition (possibly requiring data shuffling).
- *filter*: discards the elements of an RDD that do not satisfy a given condition.

As described above, our algorithm comprises a bootstrap stage, in which the initial candidate set is generated, followed by an iterative procedure mainly based on two steps: *support calculation*, computing for each candidate its support and filtering the ones not satisfying the imposed conditions; and *candidate generation*, using the n-length sequences to generate the (n+1)-length candidates.

The bootstrap step generates the initial candidates producing, for each sequence in the database, all the sub-sequences of length μ associated with their respective sequence identifier and position within the sequence.

The support computing step is reduced to the well known *WordCount* problem, easily carried out by the Spark framework, where the pair candidate - sequence id is used as key. At the end of this step, candidates that do not satisfy the minimum support condition are filtered out. If the result of the filtering is empty or the maximum length α has been reached the algorithm terminates, otherwise retained candidates are added to the results and the next step candidate generation is performed.

The aim of the Candidate Generation is to generate the next-length candidates starting from what survived the filtering process. We generate the new set of candidates as follows: if two frequent sequences were generated from the same sequence in the database and their positions are consecutive, then the join of the two subsequences is a new candidate (e.g., `ABC` and `BCD` , that appear respectively at

position 2 and 3 of the sequence AAABCDEF, generate the next candidate ABCD).

The bootstrap step of our algorithm (Step 0) loads the sequence database, stored as a text file which follows the schema *(ID, Sequence)*, and generates the initial set of candidates of length equal to the user-provided minimum length μ; this is done generating, for each sequence, the set of distinct sub-sequences of length μ associated with the ID and position in which the sub-sequence occurs in the original sequence. When the dataset is loaded into the initial RDD, it is horizontally partitioned in a number of partitions, namely p, settable by the user.

B. Execution example

For the sake of clarity, we also present the execution of our algorithm through an example on a toy dataset. We will run the algorithm on the following sequence database:

seq-id	**seq**
1	ATTTC
2	TCTTT

seeking for frequent sequences of minimum length $\mu = 2$ and minimum support $\sigma = 2$.

Bootstrap. For each sequence in the database, all the sub-sequences of length equals to μ are generated and emitted along with the sequence identifier and the list of position where they are found:

sub-seq	**<seq-id, positions+ >**
AT	<1, [0]>
TT	<1, [1, 2]>
TC	<1, [3]>
TC	<2, [0]>
CT	<2, [1]>
TT	<2, [2, 3]>

Iteration 1. The first iteration starts reducing by key (step 1) the RDD produced in the previous step. The result is an RDD in which each sub-sequence appears only once and is associated with all the sequence identifiers, and respective positions, in which the sub-sequence occurs:

sub-seq	**(seq-id, positions+)+**
AT	[<1, [0]>]
TT	[<1, [1, 2]>, <2, [2, 3]>]
TC	[<1, [3]>, <2, [2, 3]>]
TA	[<2, [1]>]

The support σ of each sequence can be trivially obtained as the number of unique identifiers associated to each entry in the previous RDD, i.e.,:

AT σ = 1
TT σ = 2
TC σ = 2
TA σ = 1

In the next step (step 2), the previous RDD is filtered discarding the entries that do not satisfy minimum support and minimum frequency, returning the RDD:

sub-seq	**(seq-id, positions+)+**
TT	[<1, [1, 2]>, <2, [2, 3]>]
TC	[<1, [3]>, <2, [2, 3]>]

Since the filtered RDD is not empty, the execution moves on with the candidate generation; step 3.1 reverts the representation of the previous RDD, generating a new RDD in which the key is the sequence identifier and the value contains the associated sub-sequences with their respective location indexes:

seq-id	**(sub-seq, positions+)+**
1	<TT, [1,2]>
2	<TT, [2, 3]>
1	<TC, [3]>
2	<TC, [0]>

Then, the RDD is reduced (step 3.2), so that its entries are aggregated by sequence identifier:

seq-id	**(sub-seq, positions+)+**
1	[<TT, [1, 2]>, <TC, [3]>]
2	[<TC, [0]>, <TT, [2, 3]>]

Within the last *flatMap* of this group (3.3), we compute the new candidates for the next iteration:

1) the list of positions represented as value in the previous RDD is flattened and elements are sorted by position index:

seq-id	**(sub-seq, positions+)+**
1	[<TT, 1>, <TT, 2>, <TC, 3>]
2	[<TC, 0>, <TT, 2>, <TT, 3>]

2) a moving window of length 2 is used to compare position indexes and a new candidate is produced for each pair of consecutive sub-sequences:

seq-id	**(sub-seq, positions+)+**
1	[<TTT, 1>, <TTC, 2>]
2	[<TCT, 0>, <TTT, 2>]

The output RDD is then flattened at step 3.3. and provided as input to the next iteration:

sub-seq	**(seqID, positions+)**
TTT	<1, [1]>
TTC	<1, [2]>
TCT	<2, [0]>
TTT	<2, [2]>

Iteration 2. The algorithm then restarts from step 1 and calculates the support value for each input sequence. Only sequence "TTT" survives to the filtering and is added to the results.

Next step produces the new candidate sequences; in this case the no pair of consecutive sequence can be joined, therefore the next candidate set is empty.

Iteration 3. The algorithm receives an empty list of candidates, therefore the convergence test succeed and the algorithm terminates returning as output all the frequent sequences identified by the previous iterations, i.e.,"TT", "TC" and "TTT". Given a database of n sequences of

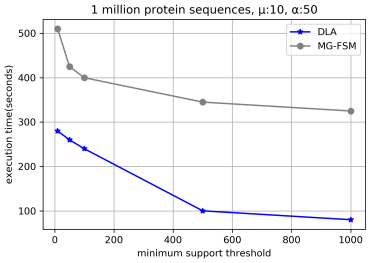

Figure 1. Execution time comparison with different minimum support threshold.

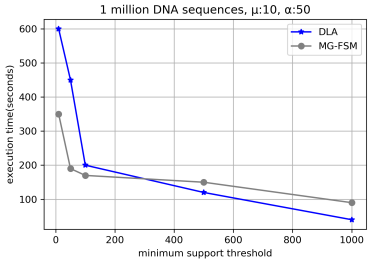

Figure 2. Execution time comparison with different minimum support threshold.

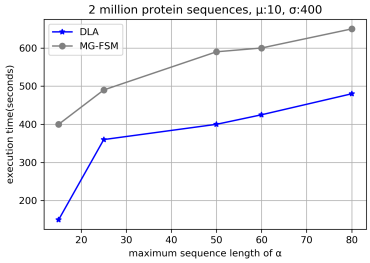

Figure 3. Execution times in function of the maximum allowed frequent sequence length α.

lenght l and fixed the query parameters σ and μ, under the assumption that $n \gg l$, the time complexity of the algorithm is $O(n)$.

IV. Experimental results and performance analysis

We have conducted extensive experiments to verify the performance of DLA with real world big data. We also compared our algorithm with an existing MapReduce method. The experiments were executed on Apache Spark 2.3.1 built on Hadoop 2.7.3 and in a fully distributed cluster environment consisting of 6 machines each one contains 5 cores of Intel(R) Xeon(R) E5-2660 2.0GHz CPU. All machines run on Ubuntu 16.04 operating system. So as to measure the performance of our algorithm, we used two real-world data sets, which were publicly provided by the National Center for Biotechnology Information(NCBI). The protein data set was extracted by the conjunction of (1) search category= "Protein", (2) species= "Bacteria", (3) organism= "Escherichia coli", and (4) release date=[2014/01, 2018/05]. The DNA data set was extracted by the conjunction of (1) search category= "Nucleotide", (2) species= "animals", (3) organism= "Homo sapiens" and (4) release date=[2011/1, 2018/05]. In Table I the main characteristics of the two datasets are reported.

Table I
Dataset characteristics

	DNA sequences	**Protein sequences**
Average length	104	365
Maximum length	200	34 366
Total sequences	5 000 000	25 000 000
Distinct items	5	21
Total bytes	586 183 393	9 250 844 714

Firstly, we tested the impact of the pruning strategy we adopted in DLA letting the minimum support σ vary. So as to give more value to our results, and provide the reader with a term of comparison, we run our algorithm against MG-FSM [11], which represent the state of the art for the problem of mining sequence pattern within *big data*. For this experiment we used the 1 million protein sequences as dataset. Figure 1 shows execution times with minimum support σ from varying from 10 to 1,000. For all the various support values DLA appears to be 10x faster that MG-FMS.

We also repeated the same experiment but using the 1M DNA dataset; in this case, as represented in Figure 2, except for low values of support where the performances of MG-FSM are way better, the two algorithms have similar execution times.This result suggests that DLA performs better than its competitors when both the alphabet is large enough and the sequences are long, thus making the adopted strategy for pruning the search space works at its best.

Besides the huge number of sequences and the various minimum support values another challenge is to achieve performance with different lengths of frequent sequences. We tested this running our algorithm of a 2M sequence database, fixing as parameters $\mu = 10$ and $\sigma = 400$ (which corresponds to the 0.02 % of the number of sequences in the database) and varying the parameter α that indicates the maximum allowed frequent sequence length. Results are reported in Figure 3: as expected, execution times increases as the maximum length grows, given that minimum support threshold remains constant. Again for the protein dataset, which contains very long sequences and on a large alphabet, our algorithm clearly out-performs MG-FSM.

In the next test we want to report how the algorithm scales with respect to the size of the of the input dataset. In order to have dataset with different sizes we took random sampling from the full protein dataset. For all of these 8 datasets we run the same experiment with the very same parameters for minimum and maximum length of the frequent sequence: $\mu = 15$ and $\alpha = 30$ and requiring minimum support σ to be equal to 0.02 of the number of regions for each of the dataset. Under these circumstances, we believe our test to be as fair as possible. Execution times are reported in Figure 4. Furthermore, we computed the **slope** as Δ of execution time / Δ of sequences size. This represents the additional execution time needed for processing 1k sequences more. As can be seen in Figure 4, the blue line representing the slop is almost constant in the seven points where we measured

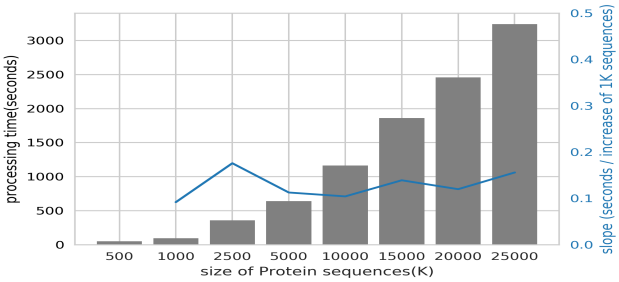

Figure 4. Execution time with different size of sequences.

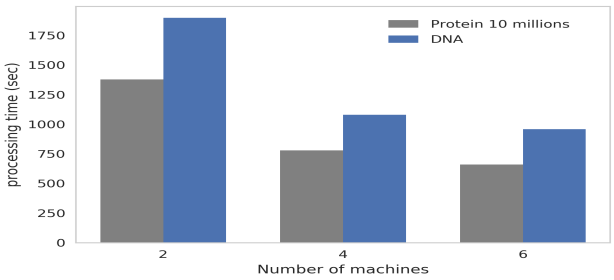

Figure 5. Execution time with different number of cores.

it. Therefore, this shows that in response of an increasing the dataset size, the algorithm scales linearly.

Finaly, we made a test to prove the scalability of DLA with respect to the number of executors in the cluster. Results are reported in Figure 5. We run DLA on two datasets 10M protein and 5M DNA sequences, represented in grey and blue respectively. In all the three runs we let invariant the execution parameters, such as $\sigma = 2000$, $\mu = 8$ and $\alpha = 15$). Results in Figure 5 show how the execution time of the algorithm decreases with the number of executors. This ensures that, provided a "big enough" cluster even, huge dataset can be analyzed in reasonable time.

V. Conclusion

In this paper, we propose a distributed method in Spark for frequent sequence mining in large scale databases. This model is effective since it will always find the pattern if the pattern exists. It reduces the search space by pruning the portion not related to the pattern. Our algorithm is proved to be efficient and highly scalable. Based on GSP algorithm, keeps its performance advantages of apriori-model and is taking advantage the new techniques of Apache Spark computing framework.

Acknowledgment

The authors would like to thank all the colleagues of the GeCo ERC project for their help.

References

[1] R. Agrawal, and R. Srikant, *"Mining Sequential Patterns"*, in Proc. of the 11th Int'l Conf. on Data Engineering(ICDE95), 1995.

[2] L. Liu and M. T. Ozsu, *"Apriori Property and Breadth-First Search Algorithms"*, Springer US, pp.124-127, 2012.

[3] R. Srikant, H. Toivonen, A. I. Verkamo, R. Agrawal, and H. Mannila, *"Fast discovery of association rules. Advances in knowledge discovery and data mining"*, American Association for Artificial Intelligence, pp.307-328, 1996.

[4] M. Zaki, *"Efficient enumeration of frequent sequences"*, Proceedings of the 7th international conference on Information and knowledge management, pp.68-75, 1998.

[5] J. Ayres, J. Gehrke, and T. Yiu, *"Sequential pattern mining using a bitmap representation"*, Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.429-435, 2002.

[6] J. Han, J. Pei, and B. Mortazavi-Asl, *"FreeSpan: frequent pattern-projected sequential pattern mining"*, Proceedings of the 6th ACM SIGKDD inter-national conference on knowledge discovery and data mining (KDD00), pp.355-359, 2000.

[7] J. Pei,J. Han, and B. Mortazavi-Asl, *"PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth"*, Proceedings of the 7th ACM SIGKDD international conference on knowledge discovery anddata mining, pp.215-224, 2001.

[8] C. C. Chen, C. Y. Tseng and, M. S. Chen, *"Highly scalable sequential pattern mining based on Mapreduce model on the cloud"*, IEEE International Congress on Big Data, pp.310-317, 2013.

[9] C. C. Chen, H. H. Shuai and, M. S. Chen, *"Distributed and scalable sequentia lpattern mining through stream processing"*, Knowledge and Information Systems, pp.365-390, 2017.

[10] Y. Wei, D. Liu, and L. Duan, *"Distributed PrefixSpan algorithm based on MapReduce."*, International Symposium on Information Technology in Medicine and Education, pp.901-904, 2012.

[11] K. Beedkar, K. Berberich, R. Gemulla, and I. Miliaraki,*"Mind the Gap: Large-Scale Frequent Sequence Mining"*, Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp.797-808, 2013.