Highly Scalable Sequential Pattern Mining Based on MapReduce Model on the Cloud

Chun-Chieh Chen[1], Chi-Yao Tseng[2], and Ming-Syan Chen[3]
[1]*Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan, R.O.C., Email: ccchen@arbor.ee.ntu.edu.tw*[1]
[2,3]*Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan, R.O.C., Email: {cytseng*[2]*, mschen*[3]*}@citi.sinica.edu.tw*

***Abstract*—Sequential pattern mining is an essential data mining technique that has been widely applied to many real-world applications. However, traditional algorithms generally suffer from the scalability problem when dealing with big data. In this paper, we aim to significantly upgrade the scale and propose Sequential PAttern Mining algorithm based on MapReduce model on the Cloud (abbreviated as SPAMC). Derived from the prior SPAM algorithm, we design an iterative MapReduce framework to efficiently generate and prune candidate patterns when constructing the lexical sequence tree. This framework not only distributes the sub-tasks of tree construction to independent mappers in parallel, but also enables the parallel processing of support counting. We conduct extensive experiments on the cloud environment of 32 virtual machines with up to 12.8 million transactional sequences. Experimental results show that SPAMC can significantly reduce mining time with big data, achieve extremely high scalability, and provide perfect load balancing on the cloud cluster.**

***Keywords*-Sequential Pattern Mining; Big Data; Cloud Computing; MapReduce framework.**

I. Introduction

In recent years, sequential pattern mining [1][2] has become an essential data mining technique and been applied to many applications, such as gene analysis in bioinformatics, customer behavior prediction, and intrusion detection of network attack. Generally, the main objective of sequential pattern mining is to discover frequent sequences within a transactional database. The problem was first proposed in [3], and the formal definition can be detailed as follows.

Definition 1: Let D be a sequence database, and $I = \{x_1, \cdots, x_m\}$ be a set of m different items. $S = \{s_1, \cdots, s_i\}$ is a sequence consisting of an ordered list of itemsets. An itemset s_i is a subset of items $\subseteq I$. A sequence $S_a = \{a_1, \cdots, a_n\}$ is a subsequence of sequence $S_b = \{b_1, \cdots, b_m\}$ where $1 \leq i_1 < \cdots < i_n \leq m$ such that $a_1 \subseteq b_{i1}, a_2 \subseteq b_{i2}, \cdots, a_n \subseteq b_{in}$. Sequential pattern mining is to find the complete set of sequential patterns whose occurrence frequencies $\geq min\_sup * |D|$, where min_sup is a minimum support threshold.

Although numerous approaches, including Apriori-based [3][4][5], projection-based [6][7][8][9], and pattern growth-based algorithms [8][9][10][11], have been proposed to mine sequential patterns, existing methods designed for running on a single machine generally suffer from the problems of limited memory and computing power when dealing with a huge amount of data. For example, AprioriAll is the first Apriori-based method and it spends more than 15,000 seconds on a 400,000 sequences dataset [12]. An improved Apriori-based algorithm GSP [5] still takes more than 700 seconds on a 400,000 sequences dataset. To further enhance the performance, researchers have proposed methods [4][5][6][7][8][9] that apply advanced data structure, design pruning mechanism, or project data into smaller space. Nevertheless, these algorithms are inherently designed to mine sequential patterns on a single machine. When dealing with a considerably large number of transactional sequences that may not be entirely loaded into main memory, most existing algorithms cannot efficiently complete the mining process.

To overcome the scalability challenge, there are several approaches targeting at parallel mining [6][12] and distributed mining [13] on sequential patterns. Huang et al. [14] first propose a sequential pattern mining algorithm on the Hadoop [15] cloud computing environment. Although the scalability problem has been partially improved by these types of methods, they still have limitations and are not highly scalable. To remedy these problems and further upgrade the scale of sequential pattern mining, in this paper, we propose the SPAMC (standing for Sequential PAttern Mining on the Cloud) algorithm based on MapReduce framework [16]. SPAMC is mainly derived from the related work SPAM [4] proposed by Ayres et al.

SPAM utilizes bitmap representation to facilitate the mining process and only requires scanning database once, significantly enhancing the efficiency performance. However, while SPAM still executes in a centralized manner, which has limited memory and computation power, the scalability of SPAM is restricted. Nevertheless, fast operation and one database scan make SPAM potential to scale significantly on the cloud. Therefore, in this paper, we address the challenging issue of sequential pattern mining on big data by designing a cloud-based algorithm SPAMC derived from SPAM.

SPAMC adopts an iterative MapReduce framework consisting of two main phases: scanning phase and mining

978-0-7695-5006-0/13 $26.00 © 2013 IEEE
DOI 10.1109/BigData.Congress.2013.48

phase. Initially, scanning phase splits the original database into several partitions and transforms the sequence database into bitmap representation. Transformed database will be saved in a distributed hash table (abbreviated as DHT) on the distributed file system. Therefore, each machine can effectively access required information from DHT, and both memory requirement and overall transmission cost can be significantly reduced.

In the mining phase, SPAMC employs iterative MapReduce steps to process the sub-tasks of parallel tree construction with numerous independent mappers. The main spirit of this design is transforming the serial processing of tree construction into parallel processing, so that sub-tasks can be distributed and executed by many machines simultaneously. In addition, the pattern checking of each candidate pattern can also be distributed to mappers, and thus one machine will not be overloaded due to limited resources.

We conduct extensive experiments with the size of sequence database up to 12.8 million sequences. Experimental results verify that SPAMC significantly outperforms existing algorithms and has good efficiency to mine sequential patterns on big data.

The rest of this paper is organized as follows. We review SPAM algorithm and survey related studies in Section II. The proposed algorithm SPAMC is described in Section III. The experimental evaluation is presented in Section IV. Finally, Section V concludes this paper.

II. Preliminaries

A. Review of SPAM

To avoid multiple database scans of Apriori-based approaches and to enhance mining efficiency, Ayres et al. proposed SPAM algorithm [4]. Initially, SPAM scans the original database once and transforms it into a vertical bitmap table. Figure 1 shows an example transactional database and the transformed bitmap table. If item $\{A\}$ appears in transaction j of customer 1, the corresponding bit is set to '1'; otherwise, the bit is set to '0'. Subsequently, the core of SPAM is the construction of a lexical sequence tree (as depicted in Figure 5) that contains all candidate sequential patterns. Each tree node represents a candidate pattern. Instead of re-scanning the database, the support counting of each node can be efficiently processed by the fast bit-AND operation. In addition, note that the tree construction follows the depth first search (DFS) traversal. During the DFS traversal, each node should be performed the sequence-extension step (S-step) and the itemset-extension step (I-step) to iteratively extend sequential patterns. Moreover, to further reduce the traversal space, pruning strategy based on Apriori principle has been applied to both S-step and I-step extensions. If the support count of node is larger than or equal to min_sup, DFS traversal continues from this node until no patterns can be generated.

Cid	Tid	Itemset
1	1	{A}
1	3	{B}
1	7	{C}
1	12	{A}
2	2	{A, D}
2	4	{B}
3	5	{B, C}
3	8	{B}
3	9	{C}
4	6	{A}
4	10	{B}
4	11	{A}

(a)

Cid	Sequence
1	({A},{B},{C},{A})
2	({A,D},{B})
3	({B,C},{B},{C})
4	({A},{B},{A})

(b)

		Item			
Cid	Tid	{A}	{B}	{C}	{D}
1	1	1	0	0	0
1	3	0	1	0	0
1	7	0	0	1	0
1	12	1	0	0	0
2	2	1	0	0	1
2	4	0	1	0	0
2	-	0	0	0	0
2	-	0	0	0	0
3	5	0	1	1	0
3	8	0	1	0	0
3	9	0	0	1	0
3	-	0	0	0	0
4	6	1	0	0	0
4	10	0	1	0	0
4	11	1	0	0	0
4	-	0	0	0	0

(c)

Figure 1. (a) transaction database. (b) sequence database D. (c) vertical bitmap representation of D.

We demonstrate I-step and S-step procedures of item $\{A\}$ in Figure 2. For I-step, we simply do the bit-AND operation on the bitmaps of $\{A\}$ and $\{B\}$ on the bitmaps of $\{A\}$ and $\{B\}$ to get the result of $\{A,B\}$. For S-step, we first need to transform the bitmap of $\{A\}$ into $\{A\}_s$. The index of the first 1 bit in $\{A\}$ should be transformd into 0. Then all the bits which are behind this bit should be set to '1'. When the transformed bitmap $\{A\}_s$ is obtained, we do the bit-AND operation on $\{A\}_s$ and $\{B\}$ to get the result of S-step of $(\{A\},\{B\})$. After I-step or S-step is finished, we accumulate the number of sequences that have more than one 'true' bits in bitmap results. As shown in Figure 2, it can be seen that the support count of $\{A,B\}$ is 0 and that of $(\{A\},\{B\})$ is 3. If min_sup is set to 50% (i.e., 2 sequences), $\{A,B\}$ will be viewed as infrequent and will be pruned. Following this procedure, SPAM can construct the whole lexical sequence tree and generate the complete set of frequent sequential patterns.

B. Related Works

We briefly review traditional sequential pattern mining techniques designed for running on a single machine, and then focus on the related studies of distributed-based and cloud-based methods. Traditional methods can be generally classified into Apriori-based [3][4][5], projection-based

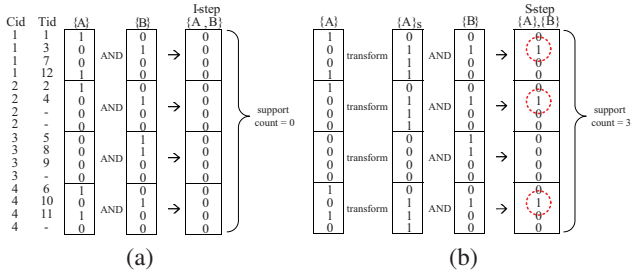


Figure 2. An example of (a) I-step and (b) S-step.

[6][7][8][9], and pattern growth-based [8][9][10][11] algorithms. Apriori-based algorithms, including AprioriAll [3], SPAM [4], GSP [5], mainly generate candidate and prune sequential patterns based on Apriori principle. Projection-based algorithms, including FreeSpan [7] and PrefixSpan [8], project the database into sub-databases and generate length-k patterns based on the length k-1 patterns without candidate generation. For pattern growth-based methods including PrefixSpan [8], SPADE [9], Pisa [11], specific data structures are usually devised to generate, maintain, prune, and extend candidate patterns. Although the performance of these algorithms has been gradually enhanced, their design is inherently suited to the single machine environment, indicating not able to efficiently deal with a huge amount of data due to limited resources.

On the other hand, numerous distributed algorithms have been proposed. They divid data into many small chunks and perform the mining process in parallel with multiple machines. Related studies include parallel sequential mining [12][13] and parallel tree-projection sequential mining [6]. In [13], the authors propose a distributed Apriori-based algorithm that processes generate-and-test in a heterogeneous cluster environment based on the block-based partition method. Zaki et al. [12] propose pSPADE algorithm based on share-memory architecture that can share database via networking. Guralnik and Karypis [6] present a tree-projection-based algorithm on distributed system that decomposes the projection tree into many partitions. Although the above algorithms have demonstrated that mining sequential patterns can be done in a distributed way, each machine has to share data through communication with one another, incurring high mining overhead.

On the other hand, some researchers attempt to design algorithms on the cloud computing environment. Hao et al. design a parallel version of the FP-Growth algorithm on the cloud, which runs parallel tasks (including counting, FP-growth, and aggregation) on multiple machines. In [17], the authors propose a parallel closed sequential mining on the cloud, which extends the sequential patterns based on forward and backward extension. DPSP [14] is the first cloud-based sequential pattern algorithm for progressive databases. In this algorithm, the input data is divided into many progressive windows and it performs the process of candidate generation on many machines independently. After the candidate patterns are generated, each machine executes the support assembling job to count the occurrence frequencies of candidate patterns. One major drawback of DPSP is that it needs to proceed through numerous rounds of MapReduces jobs, which will easily cause the load unbalancing problem and incur high cost of MapReduce initialization. In addition, this work focuses on the concept of period of interest (POI), where the database is divided into many POI windows. Thus, its ability of handling longer sequential patterns is not demonstrated. We will extensively compare the performance of SPAMC with state-of-the-art approaches in Section IV.

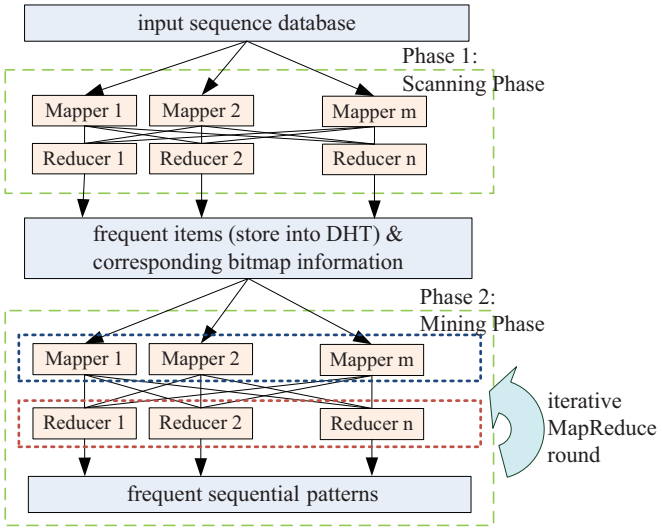


Figure 3. Framework of SPAMC.

III. SPAMC Algorithm

A. Framework of SPAMC

SPAMC is a cloud-based version of sequential pattern mining algorithm consisting of two phases: 1) scanning phase, and 2) mining phase. Figure 3 shows the framework of SPAMC and the algorithmic form of SPAMC is shown in Algorithm 1.

In the scanning phase, we achieve high performance by distributing tasks on multiple computers with a round of Mapreduce job being employed to proceed in parallel by distributing sub-tasks to independent machines. Each mapper scans and transforms a partitioned database, and reducers are in charge of counting the occurrence frequency of each item. At this stage, infrequent items will be eliminated. Finally, the bitmap information of frequent items will be stored into a distributed hash table (DHT) that can be accessed in the mining phase.

Algorithm 1: SPAMC

Input: sequence database *D*
min_sup, minimum support threshold
depth, sub-tree depth
Output: complete set of frequent sequential patterns
1: var *DHT*; // used to store the bitmap of frequent items
2: var *Candidate*; // used to store candidate patterns
3: *Candidate = ScanningJob(D, min_sup, DHT)*;
4: **while** *Candidate.size* > 0 **do**
5: *MiningJob(Candidate, min_sup, DHT, depth)*;
6: output frequent sequential patterns;
7: **end while**

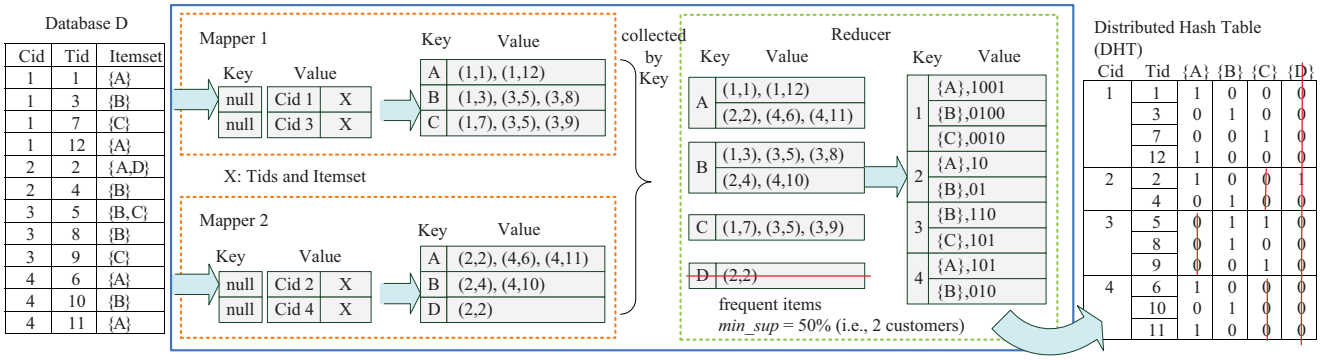


Figure 4. Demonstration of scanning phase.

Subsequently, in the mining phase, the sequential pattern mining tasks are processed in parallel by distributed machines. Each machine constructs partial sub-trees of the lexical sequence tree and produces candidate subsequences locally and independently. Note that the main job of the mining phase is to construct the complete lexical sequence tree, and then all patterns can be derived. To enable the parallel processing, we design an iterative MapReduce framework that can facilitate the candidate pattern generation. Moreover, to achieve better load balancing, we adopt a new search strategy, which carries out the steps of sequence extension and itemset extension in depth first search (DFS) manner with limited sub-tree depth. This strategy effectively improves the situation that one mapper may stand and wait for a long time. In such a context, each MapReduce round will complete two levels of lexical sequence sub-tree construction. On the other hand, reducers efficiently integrate output results from mappers and do the support counting to generate frequent sequential patterns of the current sub-tree. We will elaborate on the details of these two phases in the following sections.

B. Scanning Phase of SPAMC

To avoid the situation that big data may not be fully loaded into the main memory of one single machine and to enhance the efficiency, we design to read input database with one MapReduce round in Algorithm 2. The sequences in input database D are equally split into several partitions (two partitions) in Figure 4. Each mapper reads a set of partitioned data, and each partition will be transformed to a key-value pair $< Item, (Cid, Tid) >$, where the key is item and the value is the pair of Cid and Tid. As can be seen in Figure 4, the partition 1 containing the sequences of Cid 1 and 3 is processed on mapper 1. We take item {A} for example. In the middle key-value pairs of Figure 4, mapper 1 outputs $< \{A\}, (1, 1), (1, 12) >$ because item {A} appears in the sequences of Cid 1 with Tid 1 and 12.

For the Reduce job, the output with identical key will be sent to the same reducer. As shown in lines 4-7 of Algorithm 3, a reducer sums up the support counts of the same item and stores $item$ and $support$ into local hash sup. After finishing summing up the support counts, each reducer will build the bitmap of each item. Only the items whose support counts are larger than or equal to min_sup will be retained. Finally, reducer outputs the frequent items as $< Cid, (Item, bitmap) >$ to next MapReduce job and $< Item, (Cid, bitmap) >$ to DHT in lines 9-13. At this stage, the bitmaps of frequent items can thus be directly accessed through DHT in the following mining phase.

Algorithm 2: ScanningPhase (Mapper Side)

Input: p, a data partition
1: var p = read input data;
2: var *data* = *p.value*; // < *Cid, Tid, Item* >
3: **for** each *data* in *p* **do**
4: output < *data.Item, (data.Cid, data.Tid)* >;
5: **end for**

Algorithm 3: ScanningPhase (Reducer Side)

Input: < *key, values* >, a mapper output pair in the form of < *Item, (Cid, Tid)* >
min_sup, minimum support threshold
Output: output data in transformed format < *Cid, (Item, bitmap)* >
1: var *item* = *key*; // used to store the input key
2: var *sup* < *item; support* >; // used to store support count
3: **for** each *value* in *values* **do**
4: *sup*= calculate the support count of each item from input pairs;
5: **end for**
6: **for** each < *item, support* > in *sup* **do**
7: **if** *sup(item).support* >= *min_sup* **then**
8: var *bitmap* = create the bitmap of each frequent item;
9: output < *item, (data.cid, bitmap)* > to *DHT*;
10: output < *data.cid, (item, bitmap)* >;
11: **end if**
12: **end for**

As shown in Figure 4, we set the min_sup to 50%, and a reducer simultaneously collects the sub-results from mappers 1 and 2. The key is item {A} and the values of item {A} are (1,1),(1,12),(2,2),(4,6),(4,11) on a reducer. Then the reducer can derive that the support count of {A} is 3 since item {A} appears in the sequences of Cid 1, 2, and 4. In this case, the support count of {A} is larger than min_sup, and thus A is a frequent item. The bitmap of A will be written into DHT. As for item {D}, it will be eliminated since it is infrequent. Finally, the reducer will form the bitmaps so that the outputs related to item {A} are $< 1, (\{A\}, 1001) >$, $< 2, (\{A\}, 10) >$ and $< 4, (\{A\}, 101) >$, and they will be stored into DHT. DHT is the final outcome of the scanning phase. The complete DHT of database D is shown in Figure 4.

C. Mining Phase of SPAMC

The main concept of mining phase is to generate the candidate subsequences by splitting the huge lexical sequence tree into many sub-trees, and to create all candidate sequential patterns through traversing sub-trees in a parallel manner. Specifically, we process the mining phase by iteratively executing MapReduce rounds, and each round constructs a partial sub-tree with a pre-defined limited tree depth. This design can not only leverage the power of cloud computing, but also ensure better load balancing. The mining phase contains two main procedures: (1) mapper side: lexical sequence tree construction, and (2) reducer side: merging for support counting.

1) Mapper Side: One major contribution of this paper is that we propose a distributed version of lexical sequence tree (LST), which contains the information of all subsequences and helps SPAMC generate candidate sequenctial patterns independently with MapReduce model. Each common node represents one candidate pattern. Recall that the outputs of scanning phase are numerous $< Cid, (item, bitmap) >$ pairs. To further enhance the performance, we design that different Cid with identical candidate pattern can be sent to different mappers to process in parallel. For instance, in Figure 5, mapper 1 gets $< 1, (\{A\}, 1001) >$ and $< 3, (\{A\}, 000) >$ as input. Starting from node {A}, a sequence sub-tree $T1$ is then constructed on mapper 1 based on I-step and S-step extensions in DFS manner. Note that the tree depth of sub-tree is limited by the parameter $depth$ (set to 2 in the experiments). The main purpose of this restriction is to prevent a mapper from spending too much time processing a sub-tree, which may become a bottleneck point of the whole LST construction. This design can not only enhance the performance, but ensure better load balancing on each mapper.

After the local sub-tree is constructed, from lines 3-8 in Algorithm 4, the bit-AND operation is done in each node. If the support count of this pattern is larger than zero, the mapper will output the pair $< pattern, bit\text{-}AND\_result >$.

Algorithm 4: MiningPhase (Mapper Side)

Input: *X*, an extensible node *X*
depth, maximum sub-tree depth
1: var *local_LST*; // Lexical Sequence Tree on local machine
2: *local_LST* = construct the sub-tree whose root node is *X* and depth is equal to *depth* with I-step and S-step extensions;
3: **for** each *node* in *local_LST* **do**
4: var *bit-AND_result* = do the bit-AND operation;
5: **if** *Cal_Sup(bit-AND_result)* > 0 **then**
6: output < *node.pattern, (Cid, bit-AND_result)* >;
7: **end if**
8: **end for**

Algorithm 5: MiningPhase (Reducer Side)

Input: a set of < *pattern, (Cid, bit-AND_result)* > pairs
min_sup, minimum support threshold
Output: frequent sequential patterns
1: **for** each *pattern* **do**
2: *count* = accumulate the support count of this pattern;
3: **if** *count* >= *min_sup* **then**
4: output pattern < *pattern, count* >;
5: **if** this *pattern* is a leaf node in the current sub-tree **then**
6: output *pattern* to the extensible set *Candidate*;
7: **end if**
8: **end if**
9: **end for**

Outputted pairs will be processed and merged in the reducer side. Otherwise, if the support count is zero, the mapper will not generate any output. For example, as shown in Figure 5, each node contains $pattern$, corresponding $bitmap$, and node $depth$. We assume that $depth$ is 2, min_sup is 50%, and DHT contains frequent items {A}, {B}, and {C}. If we extend the node with item {A} in $T1$, we can obtain new common nodes, ({A},{A}), ({A},{B}), and ({A},{C}) by applying S-step to this node. Also, we can obtain {A,A}, {A,B}, and {A,C} by applying I-step to the node {A}. The bit-AND operations of ({A},{B}) and {A,B} are demonstrated in Figure 5. Note that since the local support count of {A,B} is zero, we do not need to continue the subsequent extensions from this node.

2) Reducer Side: The key task of reducers is to merge intermediate outputs from mappers and to derive the final support count of each candidate pattern. The outputs with identical pattern (which is the key) will be sent to the

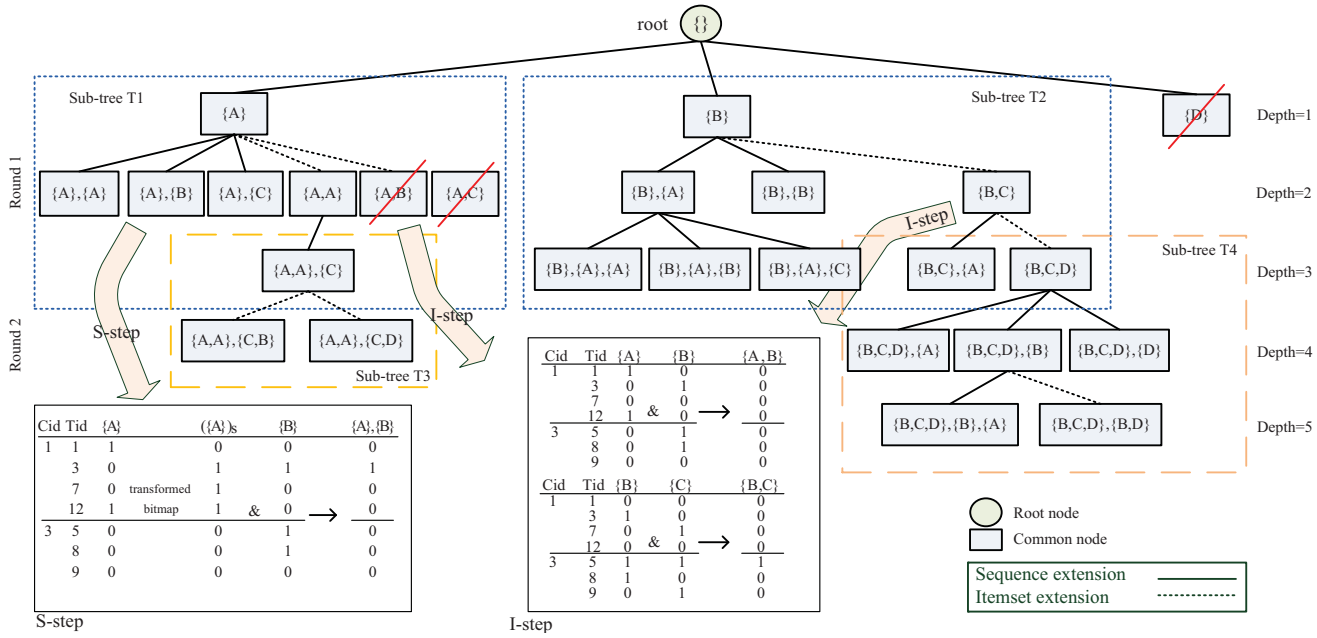


Figure 5. Demonstration of lexical sequence tree construction in mining phase (shown only partial tree here).

same reducer. As depicted in Algorithm 5, the input is in the form of $< pattern, bit\text{-}AND\_result >$ where the value field $bit\text{-}AND\_result$ is the output generated from mappers. After all candidate patterns are read, the reducer sums up the support count of each pattern by performing the bit counting on corresponding bitmap results. The final results of $< pattern, count >$ pairs of frequent sequential patterns will be outputted. Meanwhile, the patterns which are not frequent will be pruned. In addition, in lines 5-7, if a frequent sequential pattern is a leaf node in current sub-tree, this pattern will be stored into the extensible set $Candidate$. Subsequently, the mining phase initiates another MapReduce round on each pattern node in $Candidate$. Finally, SPAMC terminates when the $Candidate$ set is empty. We take the pattern ({A},{B}) in Figure 5 for example. Since there are four customers, a reducer will receive two $bit\text{-}AND\_results$ (Cid 1 and 3) from mapper 1 and two results (Cid 2 and 4) from another mapper (which is not shown).

In summary, with the design of distributed lexical sequence tree that can be constructed on mappers in parallel manner, SPAMC can achieve very high scalability. Moreover, the proposed iterative MapReduce model can also ensure better load balancing for SPAMC.

IV. Experimental Results

We have conducted extensive experiments to verify the performance of SPAMC with big data. Comparison with existing methods will be presented in Section IV-A. In Section IV-B, we will show the scalability of SPAMC. Finally, to give more insight into SPAMC, we will discuss the effects of parameters in Section IV-C.

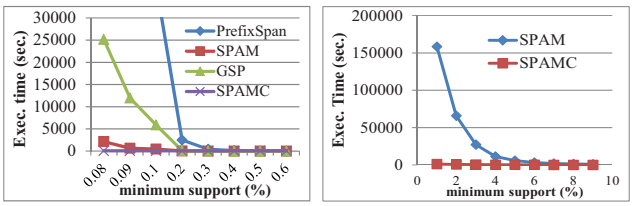
Parameter	Description
D	Number of customers
C	Average transactions per customer
T	Average items per transaction
N	Number of distinct items

Table 1. Parameters of Synthetic Datasets.

We implement SPAMC on Hadoop 1.0.3 and jdk-7u3 in a cloud environment consisting of 32 machines. One machine serves as both master and slave, and the other 31 machines are solely slaves. All the experiments are performed on machines with 2.93GHz Intel Xeon CPU, 4GB main memory, and 1GB network. The synthetic datasets used in the experiments are generated from IBM Quest data mining tool [3]. The parameters of synthetic datasets are listed in Table 1.

A. Comparison with Existing Methods

We first investigate the efficiency performance of SPAMC and several sequential pattern mining algorithms with different min_sup from 1% to 0.08% on D50kC10T8N26 dataset which contains 500,000 transactions (50,000 x 10 = 500,000). In order to compare the performance of SPAM, GSP, and PrefixSpan that are inherently designed for running on a single machine, we only test the dataset with 500,000 transactions. Note that SPAM is not able to handle large datasets due to the memory limitation on one machine. As can be seen in Figure 6(a), higher overhead (longer execution time) will be incurred when min_sup is decreased. The result in Figure 6(a) shows that SPAMC has good performance even if min_sup is low. Moreover, we can find in Figure 6(b) that SPAMC outperforms SPAM when min_sup is less than 0.07% by more than an order



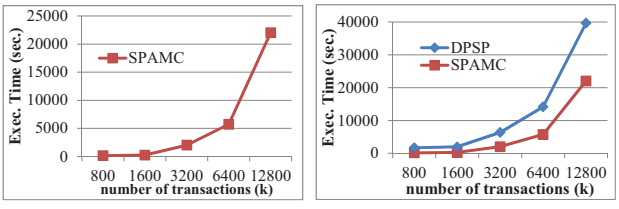
(a) (b)

Figure 6. Execution time of (a) comparative algorithms (b) SPAM and SPAMC with min_sup varied.

of magnitude. As the minimum support is lower, more and longer frequent candidate sequential patterns will be generated. Nevertheless, our algorithm can efficiently deal with huge candidate sequential patterns on many machines and reduce the memory usage by adopting the iterative MapReduce model.

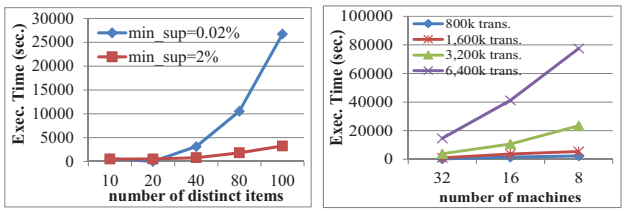
B. Scalability

To verify the capability of sequential pattern mining on big data, we attempt to execute SPAMC and DPSP [14] on the dataset with up to 12.8 million transactions. DPSP is the state-of-the-art sequential pattern mining algorithm on the cloud computing environment. We try different min_sup settings, and since all results show similar trend, we report only the results on different datasets with min_sup at 0.01%. There are 32 machines used in Figure 7, and the datasets contains from 800,000 to 12,800,000 transactions. The result shows that SPAMC indeed provides good scalability and scales nicely as we increase the number of transactions in datasets. It can be observed that the execution time rises drastically after the point 6,400,000. This phenomenon attributes to the increased memory consumption and the higher cost of network traffic. Nevertheless, SPAMC can effectively alleviate these problems by adding more machines on the cloud. Comparing with DPSP, Figure 7(b) presents that SPAMC outperforms the state-of-the-art method. Since the design of DPSP focuses on the progressive database with specific time range, when the POI (Period of Interest) becomes longer, DPSP will need more rounds of MapReduce, leading to longer execution time. It can be seen in Figure 7(b) that as the number of transactions is up to 12,800,000, SPAMC outperforms DPSP by around 50%, showing the greater scalability of the proposed method.



(a) (b)

Figure 7. (a) Scalability of SPAMC. (b) Comparison with DPSP.



(a) (b)

Figure 8. (a) Execution time with the number of distinct items varied. (b) Effect of the number of machines.

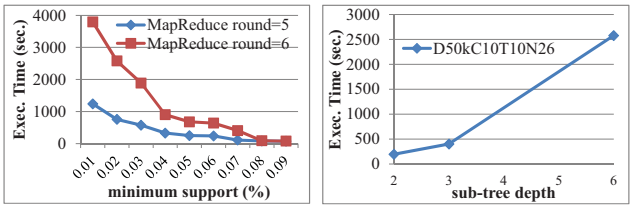
C. Sensitivity to Parameters

Several experiments are investigated to evaluate the sensitivity to parameters of SPAMC algorithm. Sensitivity analysis on many important parameters such as the number of machines, the number of distinct items, and the depth of lexical sequence tree are discussed in this section.

1) Effect of the Number of Distinct Items: We report the execution time of D50kC8T8 dataset with the number of distinct items (N) varying from 10 to 100. Figure 8(a) shows the results of $min\_sup = 2\%$ and $min\_sup = 0.02\%$. Due to the minimum data transmission and memory consumption, all the process can be executed in main memory when N equals to 10 and 20. When N grows, the execution time increases. This is mainly because SPAMC is based on SPAM, and SPAM takes more space to store the bitmap information when N is larger. Also, it takes more time to process the tree construction and bit-AND operation. In such a context, this experiment reveals that the advantage of the proposed SPAMC will be more prominent when the number of distinct items is smaller.

2) Effect of the Number of Machines: In this section, the performance evaluation of various numbers of machines is conducted. We use 8, 16 and 32 machines to perform SPAMC on four datasets with transactions from 800,000 to 6,400,000. With min_sup set to 0.01%, as shown in Figure 8(b), the execution time decreases as the number of machines is increased. Note that the execution time will be affected by the dataset characteristics, the computation complexity of mining phase, and the time spent on data transmission between machines. Therefore, the saved time will not be fully proportional to the number of machines. Nevertheless, it can be observed that more machines can greatly enhance the efficiency performance, meaning that leveraging the power of cloud is a direct way to achieve higher scalability on sequential pattern mining.

3) Effect of the Number of MapReduce Rounds: We also investigate the effect of the number of MapReduce rounds. Figure 9(a) shows the results of our proposed algorithm with different MapReduce rounds on D50kC10T10N26 dataset. As shown in Figure 9(a), SPAMC performs well in reasonable number of candidate patterns, and the execution time increases as the number of rounds increases. More MapReduce rounds indicate larger and deeper lexical sequence



(a) (b)

Figure 9. (a) Execution time of different MapReduce rounds with min_sup varied. (b) Execution time of different $depth$.

tree. Moreover, each MapReduce round incurs additional initialization cost. Therefore, this experiment demonstrates that finding longer sequential patterns also takes longer time. If only short patterns are desired, the efficiency of SPAMC can be further lifted.

4) Effect of sub-tree depth: We investigate the effect of sub-tree depth in this section. In Figure 9(b), we set sub-tree depth to 2, 3, and 6. It can be seen that as the depth increases, more time is spent on candidate pattern generation and the data transmission. Although the larger sub-tree depth indicates fewer MapReduce rounds, some mappers may take too much time dealing with a large sub-tree, which greatly affects the overall performance. To fully leverage the power of parallel processing in MapReduce model, a smaller sub-tree depth is suitable.

V. Conclusion

In this paper, we proposed SPAMC that is a highly scalable Sequential PAttern Mining algorithm on the Cloud based on MapReduce model of Hadoop framework. Overall, the contributions of this paper can be summarized as follows.

- We addressed the challenging issue of mining sequential patterns on big data in a cloud computing environment.
- We designed an iterative MapReduce framework to efficiently generate and prune candidate patterns when constructing the lexical sequence tree.
- We conducted extensive experiments on 32 virtual machines with up to 12.8 million transactional sequences, showing that SPAMC can significantly reduce mining time with big data, achieve extremely high scalability, and provide perfect load balancing on the cloud cluster.

For the future work, we plan to further investigate miscellaneous variations of sequential pattern mining on hybrid cloud environment. The topics of incremental and progressive sequential pattern mining on the cloud will also be explored in the future.

References

[1] N. R. Mabroukeh and C. I. Ezeife, "A Taxonomy of Sequential Pattern Mining Algorithms," *ACM Computing Surveys (CSUR'10)*, 2010.

[2] Q. Zhao and S. S. Bhowmick, "Sequential Pattern Matching: A Survey," *ITechnical Report*, 2003.

[3] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE'95)*, 1995.

[4] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential PAttern Mining using a Bitmap Representation," *Proc. of the 8th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD'02)*, 2002.

[5] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements," *Proc. of the 5th Int'l Conf. on Extending Database Technology (EDBT'96)*, 1996.

[6] V. Guralnik and G. Karypis, "Parallel Tree-Projection-Based Sequence Mining Algorithms," *Parallel Computing*, 2004.

[7] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining," *Proc. of the 6th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD'00)*, 2000.

[8] J. Pei, J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, U. Dayal, and M. chun Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," *Proc. of the 7th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD'01)*, 2001.

[9] M. J. Zaki, "Efficient Enumeration of Frequent Sequences," *Proc. of the 7th ACM Int'l Conf. on Information and Knowledge Management (CIKM'98)*, 1998.

[10] J. Han, J. Pei, and X. Yan, "Sequential Pattern Mining by Pattern-Growth: Principles and Extension," *Foundations and Advances in Data Mining*, 2005.

[11] J.-W. Huang, C.-Y. Tseng, J.-C. Ou, and M.-S. Chen, "A General Model for Sequential Pattern Mining with a Progressive Database," *IEEE Trans. on Knowledge and Data Engineering (TKDE'08)*, 2008.

[12] M. J. Zaki, "Parallel Sequence Mining on Shared-Memory Machines," *Journal of Parallel and Distributed Computing*, 2001.

[13] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent Itemset Mining on Graphics Processors," *Proc. of the 5th Int'l Workshop on Data Management on New Hardware (DaMoN'09)*, 2009.

[14] J.-W. Huang, S.-C. Lin, and M.-S. Chen, "DPSP: Distributed Progressive Sequential Pattern Mining on the Cloud," *Advances in Knowledge Discovery and Data Mining*, 2010.

[15] "Apache Hadoop," http://hadoop.apache.org/.

[16] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, 2008.

[17] D. Yu, W. Wu, S. Zheng, and Z. Zhu, "BIDE-Based Parallel Mining of Frequent Closed Sequences with MapReduce," *Proc. of the 12th Int'l Conf. on Algorithms and Architectures for Parallel Processing (ICA3PP'12)*, 2012.