Knowl Inf Syst (2017) 53:365–390
DOI 10.1007/s10115-017-1037-1

REGULAR PAPER

Distributed and scalable sequential pattern mining through stream processing

Chun-Chieh Chen[1,2] · Hong-Han Shuai[3] · Ming-Syan Chen[2,4]

Received: 4 July 2015 / Revised: 19 December 2016 / Accepted: 1 March 2017 /
Published online: 20 March 2017
© Springer-Verlag London 2017

Abstract Scalability is a primary issue in existing sequential pattern mining algorithms for dealing with a large amount of data. Previous work, namely sequential pattern mining on the cloud (SPAMC), has already addressed the scalability problem. It supports the MapReduce cloud computing architecture for mining frequent sequential patterns on large datasets. However, this existing algorithm does not address the iterative mining problem, which is the problem that reloading data incur additional costs. Furthermore, it did not study the load balancing problem. To remedy these problems, we devised a powerful sequential pattern mining algorithm, the sequential pattern mining in the cloud-uniform distributed lexical sequence tree algorithm (SPAMC-UDLT), exploiting MapReduce and streaming processes. SPAMC-UDLT dramatically improves overall performance without launching multiple MapReduce rounds and provides perfect load balancing across machines in the cloud. The results show that SPAMC-UDLT can significantly reduce execution time, achieves extremely high scalability, and provides much better load balancing than existing algorithms in the cloud.

Keywords Sequential pattern mining · Data mining · Cloud computing · MapReduce · Big data · Streaming MapReduce

✉ Chun-Chieh Chen
ccchen@arbor.ee.ntu.edu.tw

Hong-Han Shuai
hhshuai@nctu.edu.tw

Ming-Syan Chen
mschen@ntu.edu.tw

1 Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan

2 Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan

3 Department of Electrical and Computer Engineering, National Chiao Tung University, Hsinchu, Taiwan

4 Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

1 Introduction

In recent years, we have witnessed a large increase in the amount of data generated by Internet users. To understand the meaning of these data, it is no longer sufficient to look at individual instances of the data. Rather, it is often necessary to reason through sequences, or fragments, of information within the data. These sequences can describe a temporal effect of the data generated by the same user, or a correlation of information from various sources.

The process of uncovering important sequential patterns in a large dataset, called sequential pattern mining [25,51], has been applied in various fields, including genetic analysis, customer behavior prediction, and intrusion detection of network attacks. Data sequence discovery builds on classic algorithms, such as searching for a substring in a text file. To discover the frequent patterns, it is possible to conduct repeated searches of the file in a brute force manner, which is obviously not efficient. For better efficiency, more sophisticated algorithms, such as apriori-based [3,4,34], projection-based [14,15,30,49], and pattern growth-based algorithms [16,18,30,49], have been proposed. However, currently, the amount of data is generally several orders of magnitude more than that for which these algorithms were originally designed. Hence, typical problems such as heavy memory use and high computational load occur. As a result, modern algorithms are designed to be compatible with parallel processing in order to cope with increasing amounts of data. These methods usually start by dividing the possible pattern space into several subsets, and processing each subset on a different processor core[1] or computation server.[2] Such distributed systems are widely used framework include [1,11,14,40,50].

The previously mentioned algorithms are usually designed to discover frequent patterns only within a subset of the data. Although many useful dataset observations can be derived from these patterns, for some applications, a more global observation may be necessary, particularly when relatively long sequences tend to go beyond the boundaries of the subset. For this purpose, one can adapt these algorithms to run with the relatively large data subsets, or process the subsets in multiple passes and combine the results through standard reduction processes. These methods are clearly sub-optimal and waste a considerable amount of computational resources. Therefore, it is imperative to design an algorithm specifically for global sequence discovery in a large dataset.

One of the promising solutions, SPAM [4], performs frequent pattern mining by using vertical bitmap presentation and only requires one database scan. Moreover, [7] extended SPAM to a cloud version and focused on increasing scalability by using the iterative MapReduce model. The core value of this design is mining a sequential database using parallel processing so that sub-tasks can be distributed and executed by many machines simultaneously.

However, using iterative MapReduce model for mining frequent sequential patterns suffers from the many problems detailed in Sect. 4. One of the major problems is the unbalanced workload. Specifically, the candidate generation in mining phase may incur unbalanced workloads for different mappers since the frequency of each item is different. Therefore, most of machines may waste a lot of time in waiting for few machines, which deteriorates the performance of iterative MapReduce model. To address these challenges, we propose a cloud-based sequential pattern mining algorithm in a streaming MapReduce model, namely sequential pattern mining in the cloud-uniform distributed lexical sequence tree algorithm (SPAMC-UDLT). By using distributed message queue technique [13,22], in-memory com-

[1] OpenMP, http://www.openmp.org/.

[2] MPI, http://www.open-mpi.org/.

puting design, and streaming MapReduce model, we can improve SPAMC in many folds. Our primary contributions are as follows.

- To improve the waiting time of mappers during mining phase due to the unbalanced workloads, we modify the frequent sequential pattern mining algorithm for fitting the streaming MapReduce model instead of using the iterative MapReduce model.
- We develop a distributed streaming tree, which provides the capability of breaking the data dependence and improves efficiency of data access. In addition, the frequently used data are cached in local memory to avoid data reloading.
- The analysis on time complexity, space complexity, and waiting time is provided, which shows the advantage of SPAMC-UDLT theoretically. Moreover, the experimental results show that SPAMC-UDLT can achieve extremely high scalability and provide high speed processing in the cloud.

The remainder of this paper is organized as follows. We review the background and related work in Sect. 2. The process of sequential pattern mining is described in Sect. 3. Then, the proposed algorithm, SPAMC-UDLT, is described in Sect. 4. The experimental results are shown in Sect. 5. Finally, Sect. 6 concludes this paper.

2 Preliminary

In this section, we introduce the definition of sequential pattern mining in Sect. 2.1. The related studies of sequential pattern mining are surveyed in Sect. 2.2. Then, we explore different MapReduce models relevant to this research in Sect. 2.3.

2.1 Sequential pattern mining

Sequential pattern mining is to discover highly frequent patterns in a sequential database. The detailed formal definition is given as follows:

Definition 1 Let D be a sequence database and $I = \{x_1, \ldots, x_m\}$ be a set of m different items. $S = \{s_1, \ldots, s_i\}$ is a sequence consisting of an ordered list of itemsets, where i denotes the index of the sequence, consisting of a set of items. Itemset s_i denotes a subset of items $\in I$. A sequential pattern is defined as an ordered sequence of itemsets that frequently occurs in database D. If a sequence $S_a = \{a_1, \ldots, a_n\}$ is contained in a sequence $S_b = \{b_1, \ldots, b_m\}$, where $1 \leq n \leq m$ such that $a_1 \subseteq b_1, a_2 \subseteq b_2, \ldots, a_n \subseteq b_n$, then S_a is a subsequence of S_b. Sequential pattern mining is to find all subsequences whose occurrence frequencies $\geq min_sup$, where min_sup is the minimal support threshold.

A subsequence S_a is frequent with l items and S_a is extended to longer subsequences by appending an itemset to the end of S_a or appending a sequence with length 1 to end of S_a, where $|S_a| = l$ and $l \geq 1$. Thus, we can find a frequent sequential pattern with subsequences in database $|D|$, where the subsequence occurrences $\geq min_sup * |D|$. For the details on subsequence generation, please refer to [4].

2.2 Related works

We first review traditional sequential pattern mining techniques designed for running on a single machine and focus on the related studies of distribution-based and cloud-based methods. Traditional methods can be generally classified into apriori-based, projection-based, and pattern growth-based algorithms.

Apriori-based algorithms, including AprioriAll, GSP, and SPAM, mainly generate candidates and prune sequential patterns on the basis of the apriori principle [3,4,34]. On the other hand, projection-based algorithms, including FreeSpan and PrefixSpan, project the database into sub-databases and generate length-k patterns on the basis of the length-$(k-1)$ patterns without candidate generation [15,30]. Although the efficiency of these algorithms has been gradually enhanced, their design is inherently suited to a single machine environment and hence, they cannot deal with a large amount of data because of limited resources.

In the past decade, a variety of sequential pattern mining algorithms, such as apriori-based methods and pattern growth methods, have been developed for mining frequent patterns efficiently. In [31], the authors proposed time-aware SPAM to efficiently find temporal frequent patterns which fulfill minimal support and are within a user-defined period. On the other hand, Batal et al. [5] proposed to accelerate the process of mining frequent temporal patterns by exploiting a statistical test to effectively filter out nonpredictive temporal patterns on medical health data since a large portion of frequent temporal patterns on medical health data is irrelevant to the classification task. Moreover, ClaSP [12] exploited a vertical database representation to grow the patterns by recursively exploring the corresponding sub-trees for frequent closed patterns. For mining high utility patterns efficiently, Bai-En et al. [33] proposed a UMST framework, which adopt a MST-tree to generate mobile sequential patterns for mobile commerce applications. Luo et al. [24] proposed MSPS algorithm for mining maximal sequential patterns, which applied sampling method, prefix tree structure, and trimming process to improve the mining performance. In paper [29], SPMLS algorithm considers two constraints, which limit the maximum event length and maximum sequence length, to mine long sequences in two phases.

In addition, sequential mining can be applied in biological sequence mining, such as a protein DNA analysis in [17,21,41], which modifies PrefixSpan method to parallelly extract the frequent pattern on many machines in parallel. In DFSP [23], the authors propose a three-dimensional list for mining DNA sequences, which adopts direct access strategy and binary search to index the list structure to enumerating candidate patterns. Also, the author compares the advantage and disadvantage of bitmap presentation algorithms and projection-based algorithms in [26]. However, these works focus on efficiently mining the frequent patterns under different domain-specific constraints or on different types of data (e.g., medical database), where we focus on the general cases without any constraints. Therefore, our work can be regarded as a complement to existing works for accelerating the mining process.

Moreover, numerous distributed algorithms have been proposed; these algorithms divide data into multiple small chunks and perform the mining process in parallel with multiple machines with shared-memory environment. In paper [11], the authors propose a distributed apriori-based algorithm that processes generate-and-test operations in a heterogeneous cluster environment on the basis of the block-based partition method. Zaki et al. [50] proposed the pSPADE algorithm based on the shared-memory architecture that can share a database via networking. Guralnik and Karypis [14] proposed a tree-projection-based algorithm on a distributed system that deconstructs a projection tree into many partitions, achieving higher scalability and reducing the effect of load imbalance. Papapetrou et al. propose ACME [32], which finds frequent patterns by using an arrangement tree [28], whereas ACME applies tree structure to extract longer motif sequences on supercomputers. The previously mentioned algorithms have demonstrated that the sequential patterns can be mined in a distributed manner in a *shared-memory system*, and the local memory has to be shared among distributed machines. When the processed data are stored on another machine, the machine has to access data through communication with each other during mining phase, which increases the mining overhead. In contrast, SPAMC-UDLT adopts *distributed memory architecture* and splits the

input data into many data partitions without data dependency. Therefore, mining process can be performed on each data partition independently, i.e., mappers do not need to communicate with each other at either map phase or reduce phase during mining phase.

On the other hand, some researchers attempted to design algorithms in cloud computing. In [45], the authors propose parallel closed sequential mining on the cloud, which extends the sequential patterns on the basis of forward and backward extension. Wang et al. [42] propose PTDS, which divides transactions into smaller data partition and then applies PrefixSpan in MapReduce model. In MG-FSM [27], each machine employs FP-growth within gap constraints and adopts projection database concept in MapReduce, which includes preprocessing, partition, and mining phases for mining n-gram datasets. The authors of [19] first proposed a cloud-based sequential pattern mining DPSP for a progressive database. In this algorithm, the input data are divided into many progressive windows, and these data perform candidate generation on many machines independently. Then, reducer uses the support assembling jobs to count the occurrence frequencies of candidate sequential patterns. However, these works focus on efficiently mining the frequent patterns under different domain-specific constraints or on different types of data by using MapReduce.

The cloud-based frequent sequential pattern mining, SPAMC splits complete lexical tree into sub-trees, and machines employ iterative MapReduce model on sub-trees to generate frequent patterns. SPAMC provides higher data scalability than those of the existing distributed algorithms [7]. However, the iterative MapReduce jobs incur a performance inefficiency problem on SPAMC by reloading the intermediate results and launching the MapReduce jobs. To solve the inefficiency problem, we proposed a SPAMC-UDLT algorithm to ensure load balancing and improve the system performance in streaming MapReduce. The details of SPAMC-UDLT algorithm will be described in Sect. 4.

2.3 Highlights on MapReduce models

To the best of our knowledge, algorithm design can be especially effective when matched to a suitable model. Data mining behavior differs in mining performance in different models. In this section, we demonstrate the main distributed processing models in the cloud, i.e., MapReduce, iterative MapReduce, and streaming model, in order to obtain better algorithm design and understand different MapReduce models. Basically, MapReduce model is used to perform simple data-intensive computations and iterative MapReduce model is used to apply MapReduce to iterative algorithms. We apply the appropriate model, streaming MapReduce model, to mining frequent sequential patterns in an more effective and efficient manner.

2.3.1 MapReduce model

The MapReduce model was originally proposed at Google [9]. This model represents the data in $\langle key, value \rangle$ pair and then runs in rounds, which are composed of three consecutive phases, namely map, shuffle, and reduce. The input and output formats of MapReduce can be expressed as follows:

– Mapper: $\langle key\ input, value\ input \rangle$ to $list\langle key\ map, value\ map \rangle$
– Reducer: $\langle key\ map, list(values) \rangle$ to $list\langle key\ reducer, value\ reducer \rangle$

In the map phase, each mapper accesses the input data by a tuple at one time. After completely processing all the key-value pairs of each mapper, the algorithm generates the results in the key-value type. MapReduce will aggregate all pairs with the same key to the same reducer

in shuffle phase. Finally, the machines will process all the keys and the associated values in reduce phase.

Further, many implementations of MapReduce, such as [2,20], have been proposed. In Dryad, the authors proposed a Dryad system, which allows the developers to control jobs and data flow over a directed acyclic graph (DAG). Through the control flow of DAG, the algorithm can easily share data between machines in MapReduce.

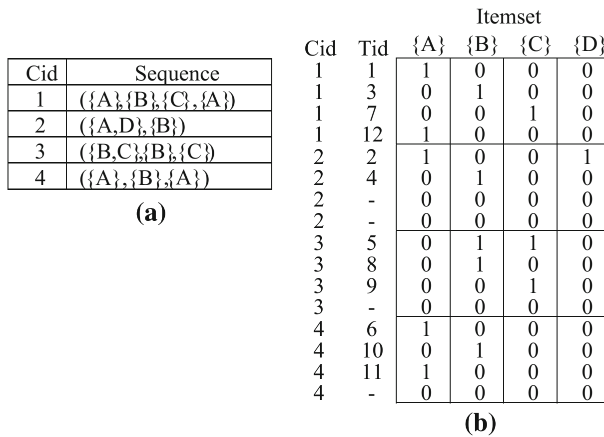
Overall, MapReduce can provide a high performance for data-intensive computing and good scalability in the case of a large amount of data, easily decompose the problem into smaller ones, and increase the number of reusable functions in a distributed environment.

2.3.2 Iterative MapReduce model

To handle naturally recursive algorithms, the data have to be loaded from extra storage and transmitted across the network in multiple MapReduce jobs. To reduce the data transmission cost, an iterative MapReduce model was proposed. In [6], the authors proposed Hadoop frameworks to serve iterative algorithms by reusing the existing Hadoop framework. To reduce the I/O cost in data access, the mappers and reducers cached the intermediate data in a local disk on each iteration. Then, with loop-aware scheduling, mappers and reducers can reload and reuse the cached data from their own local disk. To achieve a better performance, the authors proposed an extended model of the in-memory MapReduce runtime called twister [10,39]. The twister allows the worker nodes to send data directly via a broker network. The data for mappers are sent to appropriate reducers where they get cached until reducers use them in the execution. Moreover, the intermediate results of mappers are stored on a local machine. Thus, the overhead of data transmission and data loading can be reduced. In summary, the iterative MapReduce model can effectively share intermediate results between machines in each iteration and reduce the cost of relaunching MapReduce jobs by buffering data in the distributed machines.

2.3.3 Streaming MapReduce model

The streaming MapReduce model processes jobs as a series of computations in many discrete time intervals. The data are collected or loaded periodically from distributed storage. Then, the interval data are independently processed in key-value pairs on each machine. In this model, the system will repeatedly launch mappers and then launch reducer tasks in a fixed time interval. After all received data are processed by performing parallel computation, such as map or reduce, the results will write to the streaming storage. In [48], the authors proposed a discretized stream model that performs computations as a number of short tasks with a spark computing engine [47], where it can write intermediate results as the input to the next interval, and reuse data by using resilient distributed datasets (RDDs) in an iterative program. The authors [46] proposed to assist the composition of web services on large-scale data by considering the parallel technology, i.e., MapReduce framework. Compared with our proposed model, our model focuses on balancing the load of mining frequent patterns dynamically and combining both of advantage from MapReduce and streaming model together to accelerate the access of frequent pattern candidates. Recently, there have been many popular streaming frameworks including Spark, Storm, Samza, and S4 [35–38]. By exploiting the benefits of these frameworks, the machines can easily perform an iterative algorithm with one MapReduce round and share data though a string buffer with minimum system overhead.



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

(a)

		Itemset			
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

(b)

Fig. 1 **a** Sequence database D. **b** Vertical bitmap of D

3 Review on SPAM and SPAMC

This section is divided into two subsections that discuss sequential pattern mining algorithms that can mine the frequent sequential patterns by using a bitmap representation. The first subsection reviews a state-of-the-art algorithm for mining sequential patterns, and the second subsection introduces a cloud-based sequential pattern mining algorithm. The advantages of these algorithms are that they can scan a database at once and can effectively generate all the frequent sequential patterns by using a bit operation.

3.1 SPAM

To avoid multiple database scans of Apriori-based approaches and to enhance the mining efficiency, Ayres et al. proposed the SPAM algorithm [4] that utilizes the vertical bitmap data structure to achieve an efficient counting process. SPAM only requires one database scan since it transforms the original database into a vertical bitmap table as shown in Fig. 1. Specifically, all sequences are arranged in the proposed lexicographic sequence tree T, of which each node represents a candidate pattern and the root is labeled with. The construction of T follows two recursive rules: (1) if v is a node in the tree, then the children of v are all nodes v' such that v is a subsequence of v', and (2) for all nodes $u \in T$, if v' is a subsequence of u, v must be a subsequence of u. With this structure, each sequence in the lexicographic tree can be derived as (1) sequence-extended sequence, which is generated by adding a new transaction consisting of a single item to the end of its parent's sequence, and (2) itemset-extended sequence, which is generated by adding an item to the last itemset in the parent's sequence. Starting from the root node, the candidate itemsets are generated by performing the sequence-extension step (S-step) and the itemset-extension step (I-step) to iteratively extend sequential patterns with the depth-first search strategy. With the vertical bitmap data structure, support counting can be efficiently processed by a fast bit-AND operation, e.g., the Intel architecture provides 256 bits AND operation in one machine instruction.[3] We show

[3] If the bitmap vector is extremely sparse, the word-aligned hybrid code (WAH) [44] can serve for our goal. Specifically, WAH is a run-length encoding for compressing input data to words, where ANDs can be efficiently performed on any two words, and thus the bitmap representations can still work in this situation.

Cid	Tid	{A}	{B}	I-step {A , B}
1	1	1	0	0
1	3	0	1	0
1	7	0	0	0
1	12	1	0	0
2	2	1	0	0
2	4	0	1	0
2	-	0	0	0
2	-	0	0	0
3	5	0	1	0
3	8	0	1	0
3	9	0	0	0
3	-	0	0	0
4	6	1	0	0
4	10	0	1	0
4	11	1	0	0
4	-	0	0	0

(a)

{A}	{A}s	{B}	S-step {A},{B}
1	0	0	0
0	1	1	1
0	1	0	0
1	1	0	0
1	0	0	0
0	1	1	1
0	1	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	0	0
0	0	0	0
1	0	0	0
0	1	1	1
1	1	0	0
0	1	0	0

(b)

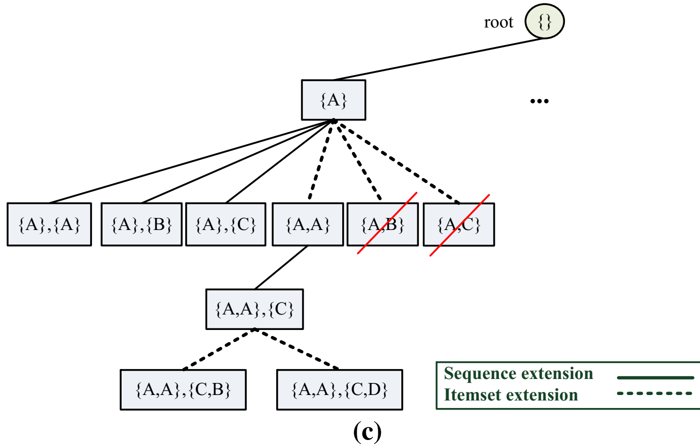


Fig. 2 An example of **a** I-step, **b** S-step, and **c** lexicographic sequence tree T

the example of bitmap information in Fig. 1 and the examples of I-step and S-step for item $\{A\}$ in Fig. 2. We set min_sup as 50%. For S-step, the bitmap representation of $\{A\}$ is first transformed to $\{A\}_s$ by setting the index of the first 1 bit in $\{A\}$ as 0 and all the bits behind this bit as 1. Then, S-step is processed by ANDing $\{A\}_s$ with $\{B\}$, and I-step is processed by ANDing $\{A\}$ with $\{B\}$. After finishing I-step and S-step, we accumulate the number of sequences that have more than one "true" bit in the bitmap results and derive that the support count of $\{\{A, B\}\}$ is 0 and that of $\{\{A\}, \{B\}\}$ is 3. After counting the support, if the support count is larger than or equal to min_sup, depth-first traversal continues until no patterns can be generated. Moreover, to reduce the search space in tree traversal, SPAM also applies pruning techniques to both S-step and I-step on the basis of the Apriori principle.

The major problem of SPAM is that the memory usage is inefficient. When the sequences become long with high frequency, SPAM takes more memory to store the bitmap of generated patterns. Furthermore, when the numbers of sequences and distinct items increase, the required space for the vertical bitmap representation also increases significantly, which hinders the capability of SPAM to mine sequential patterns on large-scale datasets on a single machine.

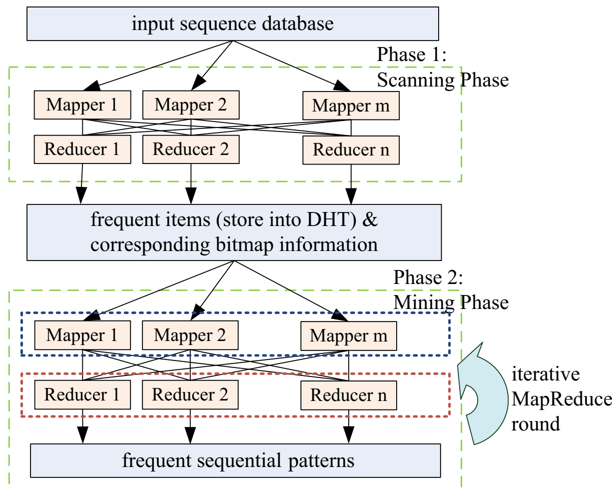


Fig. 3 SPAMC with iterative MapRedcue

3.2 SPAMC

In SPAMC, the authors attempt to make a breakthrough by designing a cloud-based mining paradigm that can significantly enhance the scalability of sequential pattern mining in an iterative MapReduce model. Figure 3 shows the framework of SPAMC, and the algorithmic form of SPAMC is shown in Algorithm 1. SPAMC is a cloud-based version of sequential pattern mining algorithm, consisting of two phases: (1) scanning phase and (2) 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**

3.2.1 Scanning phase of SPAMC

To avoid a situation in which big data may not be fully loaded into the main memory of a single machine and to enhance efficiency, SPAMC reads the input database with a MapReduce round. The sequences in input database *D* are equally split into several partitions. Each mapper reads a set of partitioned data, and each partition will be transformed into a key-value pair $\langle Item, (Cid, Tid)\rangle$, where the key is an item and the value is the pair of Cid and Tid (customer identity and timestamp). For the Reduce job, the output pairs of identical keys

are sent to the same reducer. Note that if the number of sequences is large, more reducers can be used for accelerating the reducer job. The output pairs with identical keys will be sent to the same reducer. After accumulating the support counts of all items, 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, the reducer outputs the frequent items as $\langle Cid, (Item, bitmap)\rangle$ pairs and outputs these items as $\langle Item, (Cid, bitmap)\rangle$ to a distributed hash table (DHT).

3.2.2 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 by traversing the sub-trees in a parallel manner. In particular, SPAMC carried out mining phase by iteratively executing the MapReduce rounds, and each round constructs a partial sub-tree with a pre-defined limited tree depth. Mining phase contains two main procedures: (1) mapper side: lexical sequence tree construction and (2) reducer side: merging for support counting.

(1) Mapper Side: The sub-tree construction of a lexical sequence tree (LST) is designed for parallel processing. LST contains the information of all subsequences and helps SPAMC generate candidate sequential patterns independently on MapReduce. Each node represents one candidate pattern. Recall that the outputs of scanning phase are numerous $\langle Cid, (item, bitmap)\rangle$ pairs. Each input with the same Cid is sent to the same mapper and is inserted into the local LST.

To generate candidate subsequences, SPAMC performs the modified version of sequence-extension step and itemset-extension step of SPAM on a distributed LST. The extension process with various nodes at the same tree depth can be distributed and processed in idle mappers. Further, a node may be assigned more than one customer data within each bitmap to perform candidate generation. Note that it is possible that a node parallelly utilizes a bit-AND operation on a bitmap with different Cid data. Being irrelevant to one another, different nodes and received data can be processed distributively. After constructing all sub-trees on mappers, SPAMC extends candidate subsequences by running the I-step and S-step of SPAM on all nodes of sub-trees. SPAMC sets the $depth$, which is the maximum depth of sub-trees, such that the running time of mining phase can be bound by a limited depth. If there are any nodes whose node depth is less than the maximum sub-tree depth, there are new candidate sequential patterns that can be generated. Thus, for these nodes, mappers repeatedly perform the I-step and S-step until DFS traversal of all nodes is completed or no candidate sequential patterns are generated. Then the mapper outputs the newly candidate sequential patterns as $\langle sequence, (Cid, bitmap)\rangle$ to the merging step.

(2) Reducer Side: The key task of reducers is to merge candidate subsequences generated from sub-trees on more than one machine. Reducers merge the local results from different mappers, and the output with identical subsequences is sent to the same reducer. The input is in the form of $\langle sequence, (Cid, bitmap)\rangle$, where the value field consists of Cid and bitmap in each sub-tree. All candidate sequential patterns are read. Then, the reducer summarizes the support of each candidate subsequence by directly performing the bit counting on the corresponding bitmap. In order to reduce both time and space complexity, SPAMC proposes a global view of pruning that removes any infrequent candidate subsequence. If the depth of a candidate subsequence is equal to the limited depth of this sub-tree and the count is larger than or equal to min_sup, an increasing number of new candidate sequential patterns may be

generated. Then, the candidate subsequence is put into the candidate set that is an input of the next round of MapReduce. If the candidate subsequence is frequent in the global database, the reducer outputs this candidate subsequence to the final results in $\langle sequence, count \rangle$ pairs, where count is the occurrence frequency of the sequential patterns. Then, SPAMC algorithm performs new candidate generation and iteratively performs MapReduce jobs until the candidate set is empty.

Although the abovementioned SPAMC can find frequent sequential patterns in large dataset, it is surfer from load unbalanced problem as it induces a huge overhead in long waiting time in mining phase. Also, it is not suitable to be implemented in a MapReduce framework as it induces a huge cost in data reloading when a MapReduce job is relaunched.

Algorithm 2: SPAMC-UDLT

Input: sequence database *D*
min_sup, minimal support threshold
Output: complete set of frequent sequential patterns
1: var *DHT*; // used to store the bitmap of frequent items
2: var *UDLT*; // Used to store candidate in streaming
3: *UDLT= ScanningPhase(D, min_sup, DHT)*;
4: *MiningPhase(UDLT, min_sup, DHT)*;
5: output frequent sequential patterns

4 SPAMC-UDLT algorithm

To efficiently discover the sequential patterns, we propose a new cloud-based sequential pattern mining algorithm adopting streaming MapReduce model, namely SPAMC-UDLT, with its two-phase framework shown in Fig. 4. The first phase, i.e., scanning phase, reads the input database with one MapReduce round and outputs the frequent item with the bitmap representation, while the second phase, i.e., mining phase, generates the candidate subsequences

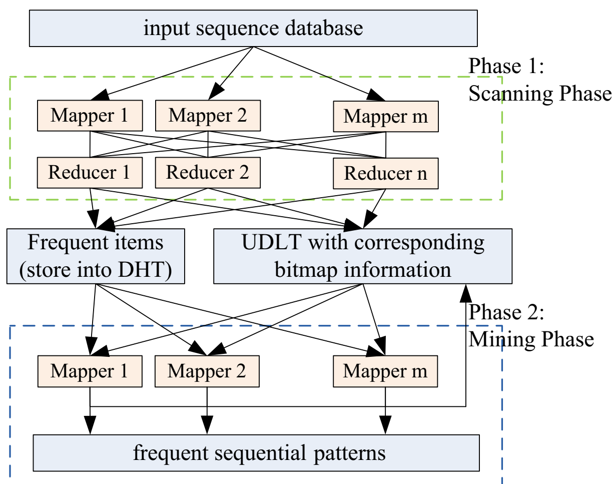


Fig. 4 SPAMC-UDLT with streaming MapReduce model

and outputs the frequent sequential patterns. There are three key challenges that exist in the design of SPAMC-UDLT, which are listed in the following paragraphs.

C1. Data reloading. Because the results of scanning phase (frequent items and the bitmap representations) will be accessed multiple times in mining phase, a new technique is required for minimizing the reloading effort.

C2. Unbalanced workload. The candidate generation in mining phase may incur unbalanced workloads for different mappers since the frequency of each item is different. For example, given 2 mappers and 2 items (A and B), one basic solution is to use one mapper for mining all the frequent sequential patterns related to A and the other mapper for B. However, the number of frequent sequential patterns related to A may be much greater than that related to B. In this case, the mapper for B wastes a lot of time in waiting for the mapper for A. It is a challenge to minimize the waiting time of each mapper for maximizing the speed of frequent sequential pattern mining.

C3. No communication allowed between mappers. In mining phase, mappers cannot directly communicate with each other due to the natural property of MapReduce, which has been designed intentionally to make sure that reliability of each map task is governed independently by the reliability of the machine. In our case, we can only gather the generated candidates in reducer, which increases the dependency between each task. It is a challenge to break this dependency in order to improve efficiency.

We address the first challenge in scanning phase and the latter two in mining phase.

4.1 Scanning phase of SPAMC-UDLT

Given the input database, the goal of scanning phase of SPAMC-UDLT is to generate frequent items with their bitmap representations and then store them in an efficient way so as to be accessed by mining phase. First, in order to increase the data scalability for large-scale datasets and reduce the waiting time induced by distributed file system, we adopt the streaming MapReduce model, in which the input data are transformed into a streaming queue. Afterward, SPAMC-UDLT distributes the transactions to each mapper with an almost equal amount of data from the streaming queue Q, and each mapper stores the data into the local memory. Afterward, SPAMC-UDLT distributes the transactions to each mapper with an almost equal amount of data from the streaming queue Q, and each mapper stores the data into the local memory. Since the output types of the Map should match the input types of the Reduce, the key-value pairs are designed as $\langle Item, (Cid, Tid)\rangle$ for calculating the frequency of each item. Therefore, the mapper takes as input $(Cid, Tid, Item)$ and outputs a key-value pair, $\langle Item, (Cid, Tid)\rangle$, to the reducers. Figure 5 shows an illustrative example of scanning phase. Eight transactions are assigned to mapper 1 and then stored in the local memory. For item $\{C\}$, mapper 1 outputs the key-value pairs as $\langle C, (1, 7)\rangle$, $\langle C, (3, 5)\rangle$, and $\langle C, (3, 9)\rangle$. After all the data are loaded, the mappers transform the input data to key-value pairs to reducers.

Intuitively, all outputted pairs with the same key are sent to the same reducer. Then, the reducer counts the frequency of the received pairs, with each reducer taking the input pairs and summarizing the frequency of distinct items. After accumulating the support counts of all the items, each reducer constructs a bitmap of each frequent item. Since the supports of all items and the corresponding bitmap will be accessed by mappers in mining phase multiple times (the first challenge), we cache the information in the local memory so that it can be efficiently accessed by different mappers. Specifically, we use a distributed hash table

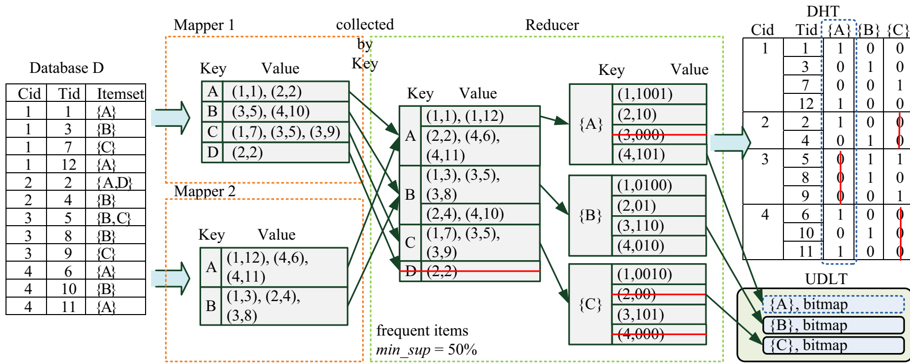


Fig. 5 Example of scanning phase

(denoted as DHT) to store this shared common information in the cloud. If the occurrence of some items is larger than the minimum support, these items are defined as frequent, and written as $\langle Item, (Cid, bitmap)\rangle$ to DHT. The detailed algorithm of scanning phase is shown in Algorithms 3 and 4.

Algorithm 3: ScanningPhase of SPAMC-UDLT (Mapper Side)

Input: *D*, a sequence database
1: var *Buffer // used to store input data*
2: **while** *D.size* > 0 // not empty
3: *buffer* = read input data from *D*;
4: var *data* = *buffer.value*; // < *Cid, Tid, Item* >
5: **for** each *data* in *buffer* **do**
6: output < *data.Item, (data.Cid, data.Tid)* >;
7: **end for**
8: **end while**

Algorithm 4: ScanningPhase of SPAMC-UDLT(Reducer Side)

Input: < key, values >, a mapper output pair in the form of < Item, (Cid, Tid) >
min_sup, minimal support threshold
Output: output data in transformed format
< *Cid, (Item, bitmap)* >
UDLT, an UDLT in streaming queue
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 *UDLT*;
10: output < *data.Cid, (Item, bitmap)* > to *DHT*;
11: **end if**
12: **end for**

4.2 Mining phase of SPAMC-UDLT

In mining phase, the goal is to efficiently generate candidate frequent sequential patterns and frequent sequential patterns in a parallel manner. However, the workloads for different machines are usually unbalanced since the frequencies of items are different. Hence, most of the machines may be idly waiting for the machines with high workloads due to the unbalanced workloads. To balance the workloads of different machines when generating candidate patterns, we propose to adopt the streaming MapReduce model to address the second challenge. Moreover, we propose a shareable uniform distributed lexical sequence tree (UDLT) in a streaming form to store the candidate frequent sequential patterns to address the third challenge, because the intermediate results of candidate frequent sequential patterns can be transmitted in a streaming form without running reducers in UDLT. Moreover, since UDLT overcomes the limitation of data accessibility in mappers, it also facilitates the operation of the streaming MapReduce model.

Specifically, UDLT starts with a root node. After generating a candidate frequent item, we insert a data node into UDLT with the frequent candidate item and its corresponding bitmap. Our goal is to enable the sequence extension and itemset-extension processes for each data node of UDLT to be performed independently so as to further generate frequent candidates. Therefore, UDLT is implemented as a distributed queue Q and the data node can be accessed as an element in Q. That is, by using distributed message queue implementation in Kafka [13,22], the data nodes are stored in distributed machines. When there are no data nodes in the local storage, the machines can directly access other machines via networking. Therefore, UDLT can achieve a better load balance and improve the overall access performance by increasing the data locality. Through the proposed UDLT, each machine can independently read the data node in distributed queue Q. Moreover, the machines only need to run the mapper procedure since UDLT contains the information of the current subsequences. Therefore, we can independently generate candidate sequential patterns on each machine.

Figure 6 and Algorithm 5 show the flowchart and pseudocode of mining phase, respectively. Recall that the outputs of scanning phase are sequences of $\langle Item, (Cid, bitmap)\rangle$ in UDLT, and DHT has been already cached on local memory in scanning phase. First, each mapper reads an input by a key $(Item)$ from UDLT in a random order and stores it to the local

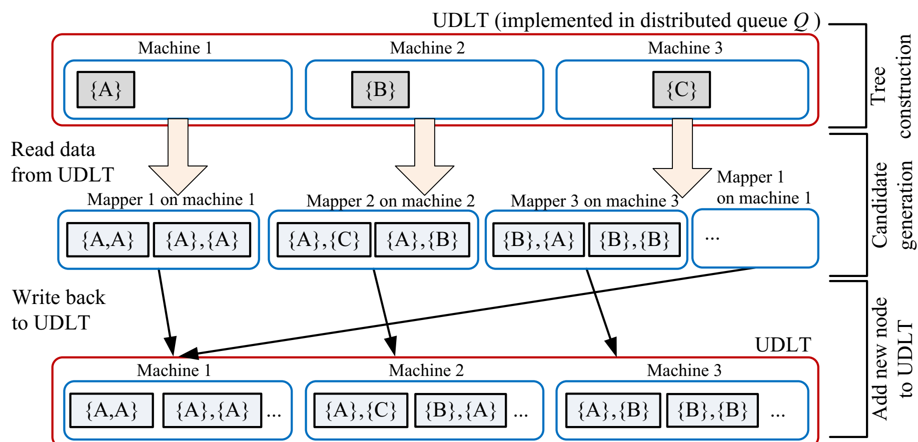


Fig. 6 Flowchart of mining phase of SPAMC-UDLT

memory in line 5 of Algorithm 5. Then, it uses the breadth-first search strategy on UDLT to discover all frequent sequential patterns.

Algorithm 5: MiningPhase of SPAMC-UDLT (Mapper Side)

```
Input: UDLT, An UDLT in streaming queue
       DHT, The cached frequent items
Output: frequent sequential patterns
1: var local_LST; // Lexical Sequence Tree on
                     Local machine
2: var freq_items = DHT // frequent items
3: var X; // an extensible node X at UDLT
4: while UDLT.size > 0
5:     X = read frequent candidates from UDLT;
6:     local_LST = construct sub-tree whose root
                     node is X with I-step and S-step
                     extensions;
7:     For each node in local_LST do
8:        var bit-AND_result = do the bit-AND
                                operation with freq_items;
9:        if Cal_Sup(bit-AND_result) > 0 then
10:          output < node.Itemset,
                        (node.Cid, bit-AND_result) >
             to UDLT and final_result;
11:       end if
12:    end for
13: end while
14: write final_result to HDFS;
```

To discover the frequent sequential patterns efficiently, we adopt a bitmap representation of sequences (lines 6 and 8 of Algorithm 5). The advantage of the bitmap representation is that it can find the occurrence of itemsets by ANDing operations, which is computationally efficient. There are two steps for generating the candidate frequent sequential patterns: sequence-extension step (S-step) and itemset-extension step (I-step). Let $I_{l,i}$ denote the i-th frequent itemset with the length l. In S-step, given a bitmap representation of a processed itemset $I_{l,i}$ and the bitmap of the j-th frequent item, i.e., $bmp_{I_{1,j}}$, we transform $bmp_{I_{l,i}}$ into a new bitmap $bmp'_{I_{l,i}}$ by setting the first 1-bit of $I_{l,i}$ as 0 and all the bits behind this bit as 1. By ANDing the transformed bitmap $bmp'_{I_{l,i}}$ with each frequent item bitmap $bmp_{I_{1,j}}$, we obtain the new bitmap of the candidate itemset $\{\{I_{1,i}\}, \{I_{1,j}\}|\forall j\}$. On the other hand, in I-step, by ANDing $bmp_{I_{l,i}}$ with each frequent item bitmap $bmp_{I_{1,j}}$, we obtain the new bitmap of the candidate itemset $\{\{I_{1,i}, I_{1,j}\}|\forall j\}$.

After finishing the S-step and I-step on the processed itemset $I_{l,i}$, all candidate itemsets generated by the itemset $I_{l,i}$ are obtained. By counting the number of 1 bits in $bmp_{\{\{I_{1,i}\},\{I_{1,j}\}\}}$ and $bmp_{\{\{I_{1,i},I_{1,j}\}\}}$, we obtain the support of candidate itemsets for pruning. If the support of itemsets is greater than the minimum support, the candidate is marked as a frequent itemset and is outputted as an $\langle Itemset, (Cid, bitmap)\rangle$ pair to UDLT (lines 9 to 10). In addition, when mappers finish the I-step and S-step process of the received nodes, mappers also write the frequent pattern in format $\langle Itemset, occurrences\rangle$ to the final results, which is located at distributed file system (HDFS). The mining process runs repetitively until all nodes in UDLT are traversed and no more frequent candidate itemsets are generated.

Example 1 Mining phase of SPAMC-UDLT.

Take Fig. 7 as an example. After scanning phase, SPAMC-UDLT outputs $\{A\}$, $\{B\}$, and $\{C\}$ to the queue of UDLT as frequent items. We assume that mapper 1 gets $\langle\{A\}$, ((1, 1001),

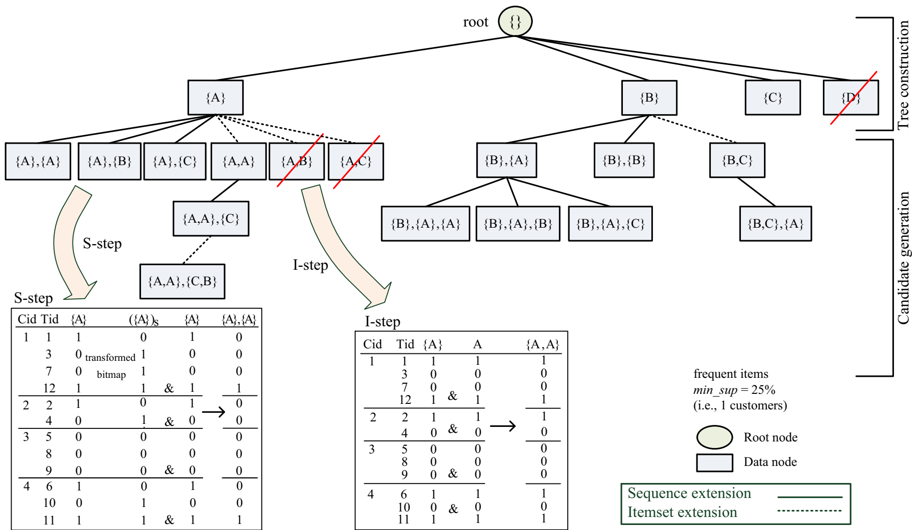


Fig. 7 UDLT of the example database in mining phase

(2, 10), (4, 101))⟩ and mapper 2 gets ⟨$\{B\}$,((1, 0100), (2, 01), (3, 110), (4, 010))⟩ as the input. Both tree data nodes $\{A\}$ and $\{B\}$ are removed from the queue of UDLT. After this, I-step and S-step are performed on data nodes $\{A\}$, $\{B\}$, $\{C\}$ to generate candidate itemsets. The dashed lines and bold lines represent S-step and I-step, respectively. Using item $\{A\}$ as an example of S-step and I-step, the bitmap of item $\{A\}$, i.e., $bmp_{\{A\}}$, is represented as 100110000101, and the transformed bitmap of item $\{A\}$, i.e., $bmp'_{\{A\}}$, is represented as 011101000011. Next, we perform ANDing for $bmp'_{\{A\}}$ and $bmp_{\{A\}}$ with each frequent item bitmap in S-step and I-step, respectively. In S-step, after the ANDing operation, we obtain 000100000001 as the bitmap of candidate itemset $\{\{A\}, \{A\}\}$. The 4th and 12th bits of itemset $\{\{A\}, \{A\}\}$ for Cid 1 and 4 are both 1, which implies that the itemset $\{\{A\}, \{A\}\}$ appears in transactions 11 and 12. Therefore, the number of occurrences of itemset $\{\{A\}, \{A\}\}$ is 2, which is equal to $min_sup = 2$, and thus itemset $\{\{A\}, \{A\}\}$ is a frequent itemset. Mapper 1 outputs node $\{\{A\}, \{A\}\}$ to UDLT. In I-step, we perform Anding on $bmp_{\{A\}}$ and $bmp_{\{A\}}$, and obtain the bitmap of itemset $\{\{A, A\}\}$ as 100110000101, which implies that the occurrence of $\{\{A, A\}\}$ is 5. Therefore, the occurrence of itemset $\{\{A, A\}\}$ is large than $min_sup = 2$, and Mapper 1 also outputs $\{\{A, A\}\}$ to UDLT. Meanwhile, mappers 2 and 3 also perform S-step and I-step on itemset {A} with frequent items {B} and {C}, respectively. The mappers consequently use a breadth-first search to traverse UDLT to discover all frequent candidates. After completing generation of candidate itemset, we obtain candidate itemsets $\{\{A\}, \{A\}\}$, $\{\{A\}, \{B\}\}$, $\{\{A\}, \{C\}\}$, $\{\{A, A\}\}$, $\{\{A, B\}\}$, $\{\{A, C\}\}$, which are the child nodes of node $\{A\}$. Similarly, we obtain candidate itemsets $\{\{B\}, \{A\}\}$, $\{\{B\}, \{B\}\}$, $\{\{B, C\}\}$ for node $\{B\}$. Finally, mappers write these frequent candidate itemsets to the final results. The mining process runs repetitively until all nodes in UDLT are traversed and no more frequent candidate itemsets are generated.

4.3 Discussions

We first show an example of which SPAMC performs transaction database poorly. Afterward, we compare the search strategies of SPAM and SPAMC-UDLT.

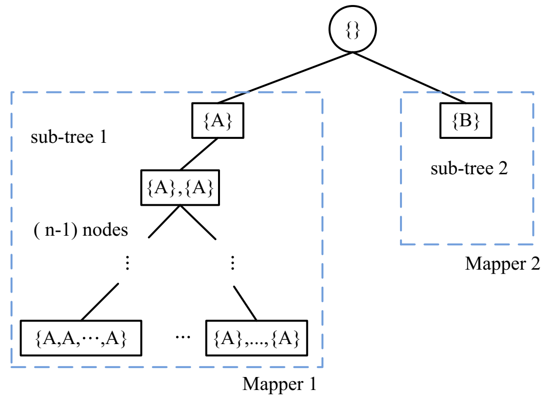


Fig. 8 Example of data skew and load inbalance

Example 2 We show a transaction database where SPAMC performs poorly.

Consider Fig. 8 as an example, where the minimum support is 2, the number of machines is m, the number of sequence is n, and the number of distinct items is 2. The mining phase of SPAMC sends $\{A\}$ and $\{B\}$ to mapper 1 and mapper 2, respectively. Afterward, item $\{A\}$ generates the sub-tree 1 on mapper 1 and item $\{B\}$ generates the sub-tree 2 on mapper 2. Mapper 1 generates $n-1$ frequent itemsets and mapper 2 only generates 1 frequent itemset. Because SPAMC adopts the iterative MapReduce model, the mapper 2 wastes a very long time until the mapper 1 finished. The waiting time is dominated by the running time of mapper 1, i.e., the time for processing $n-1$ tasks.

In contrast, the proposed SPAMC-UDLT adopts the streaming MapReduce model. Each mapper reads a candidate frequent itemset and outputs the generated candidate frequent itemsets. Note that UDLT is implemented by using distributed queue technique. Therefore, UDLT data nodes can be accessed by any distributed machines. As shown in the bottom of Fig. 8, $\{A\}$ is accessed by mapper 1 and $\{B\}$ is accessed by mapper 2. When mapper 2 finished the mining process on $\{B\}$, it will read another candidate frequent itemset, $\{\{A\}, \{A\}\}$, which is generated by mapper 1 and stored to UDLT. Therefore, the waiting time of SPAMC-UDLT is $\lceil \frac{n}{m} \rceil$ tasks, and thus the ratio of waiting time between SPAMC and SPAMC-UDLT is $\lceil \frac{n}{m} \rceil/(n-1)$, which is close to m for a large n.

In the following, we compare the search strategies of SPAM (sequential) and SPAMC-UDLT (parallel). Specifically, SPAM adopts the Depth-First Search (DFS) strategy on a lexicographic tree for generating candidate frequent patterns. The advantage of adopting the DFS strategy is that the memory usage can be minimized because only the information of the parent nodes is required. However, as shown in Example 2, when adopting the DFS strategy in distributed systems (i.e., SPAMC), it suffers from the data unbalancing problem in generating candidate frequent patterns since the heights of sub-trees for the extended sequences are usually different. As such, the machine for the shortest extended sequence idles and waits until the machine for the longest extended sequence finished its job. Moreover, when applying to large-scale datasets, each distributed machine needs to access the required global information, i.e., frequent item and corresponding bitmap information, which incurs additional costs in data communication and synchronization between distributed machines.

In contrast, SPAMC-UDLT adopts a Breadth-First Search (BFS) strategy. SPAMC-UDLT first performs S-step and I-step to extend all candidate frequent sequences for a UDLT node, and each machine is then assigned to count the support only for a candidate frequent sequence. Therefore, the load is balanced since the computation time for each machine is

Table 1 Parameters of synthetic datasets

Parameter	Description
D	Number of sequences (each sequence presents a customer data)
C	Average transactions per sequences
T	Average number of items per transaction
N	Number of distinct items
S	Average length of sequences
$max\|S\|$	Maximum length of sequence patterns

close. However, the major cost for parallelizing the UDLT with BFS strategy on distributed machines is that SPAMC-UDLT requires saving the frequent items and its corresponding bitmaps, and distributing this information to all the machines since the UDLT nodes require this information for generating new candidate frequent sequential patterns. To reduce the cost, SPAMC-UDLT further utilizes data caching and broadcasts the used information. For details on algorithm analysis of SPAMC-UDLT, please refer to online appendix [8].

5 Performance evaluation

In this section, we evaluate the performance of SPAMC-UDLT on the cloud computing platform. We have conducted extensive experiments to verify the performance of SPAMC-UDLT with big data. The dataset and environment are presented in Sect. 5.1 and Table 1. The other experimental results are as follows:

5.1 Datasets and system environment

We implement SPAMC-UDLT on Hadoop 1.2.1, Spark 0.8.1, kafka 0.7.2, 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 2.93-GHz Intel Xeon CPU, 4-GB main memory, and 1-GB network. The synthetic datasets used in the experiments are generated using the IBM Quest data mining tool [3] and the real dataset used in [43]. The parameters of synthetic datasets are listed in Table 1.

5.2 Comparison with existing methods on single machine

We first investigate the performance efficiency of SPAMC-UDLT and several sequential pattern mining algorithms with different min_sup from 0.6 to 0.08% on a dataset containing 500,000 sequences each with an average of 128 transactions ($C = 128$, $S = 8$, $N = 26$, and $T = 20$). In order to compare the performance of SPAM, GSP, and PrefixSpan, which are inherently designed for running on a single machine, we only test the smaller dataset. Note that SPAM cannot handle large datasets because of the memory limitation on one machine. As can be seen in Fig. 9a, a higher overhead (longer execution time) is incurred when min_sup is decreased. As shown in Fig. 9b, SPAMC-UDLT performs well, even if min_sup is low. As the minimum support is lower, more and longer frequent candidate sequential patterns will be generated. In this case, the overall execution time of SPAMC-UDLT is close to that of SPAMC when the dataset is small. Moreover, we can infer from Fig. 9b that SPAMC-UDLT outperforms the SPAM by more than one order of magnitude when min_sup is less than 3%.

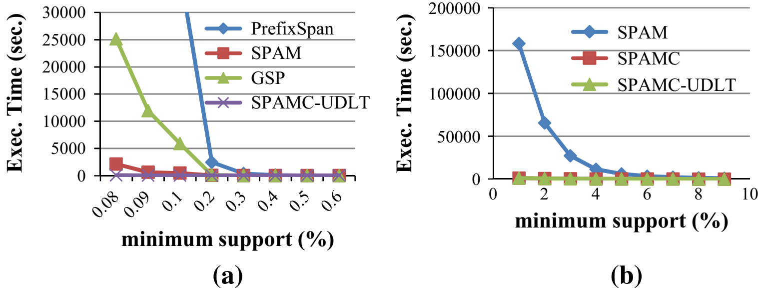


Fig. 9 Execution time of **a** comparative algorithms **b** SPAM, SPAMC, and SPAMC-UDLT with varied *min_sup*

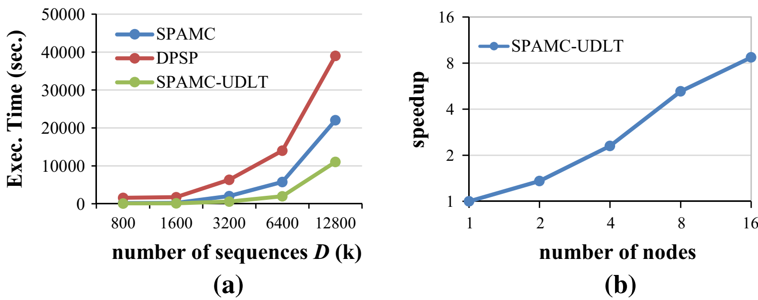


Fig. 10 **a** Scalability of SPAMC-UDLT, SPAMC and DPSP. **b** Speedup

5.3 Scalability

To verify the capability of sequential pattern mining in the cloud, we attempt to execute SPAMC-UDLT, SPAMC, and DPSP on datasets containing upto 12.8 million transactions with $S = 8$, $T = 20$, $N = 26$ and $C = 128$. We try different *min_sup* settings, and because all results show a similar trend, we report the results on different datasets with *min_sup* at 0.01%. For the illustrations shown in Fig. 10a, 32 machines are used, and the number of sequences D in the datasets were in the range 800,000 to 12,800,000 (average 128 transactions per sequence). The results show that SPAMC-UDLT provides good scalability and scales nicely as we increase the number of transactions in the datasets. Comparing the proposed algorithm with SPAMC, we have found that the execution time increases drastically after the 6400k benchmark. This phenomenon is attributed to increased memory consumption and higher network traffic cost. In DPSP, which focuses on a progressive database with a specific time range, when the number of transactions increases, more rounds of MapReduce are implemented, leading to a longer execution time.

5.4 Speedup of SPAMC-UDLT

We evaluate *Speedup* of SPAMC-UDLT on a medium-size dataset which contains 800,000 customers. Each customer is associated with 64 transactions, and there are 26 distinct items. Each machine runs on 4GB memory and 2.4GHz CPU core. The minimum support, i.e., *min_sup*, is set as 0.7%. Let T_m denote the running time of SPAMC-UDLT for m machines.

Speedup is defined as follows.

$$Speedup = \frac{T_1}{T_m}. \quad (1)$$

Figure 10b shows *Speedup* with different number of machines. As the number of machines increases from 1 to 16, *Speedup* increases linearly. *Speedup* does not linearly increase from 16 to 32 because the size of the dataset is not large. In other word, the data size in each machine is too small to show the superiority of SPAMC-UDLT since the communication cost contributes more running time than the computations.

5.5 Real dataset

We have additionally compared SPAMC-UDLT with MG-FSM on a real dataset, i.e., Twitter dataset [43]. Twitter dataset includes 12,053,495 tweets, 510,603 users, and 1,434,862 distinct items. The gap of MG-FSM is set as 0 for a fair comparison. Moreover, min_sup is ranged from 0.0033 to 0.00005%.

Figure 11a, b presents the execution time of SPAMC-UDLT, SPAMC, and MG-FSM on Twitter data, where the maximum length of frequent pattern, i.e., $max_{|S|}$, is set as 4 and 12, respectively. All results show that SPAMC-UDLT outperforms other baseline algorithms. As the length of frequent patterns increases or the minimum support decreases, the advantages of SPAMC-UDLT are highlighted.

Specifically, Fig. 11a shows that SPAMC-UDLT significantly outperforms SPAMC and MG-FSM in terms of the execution time. This is because the initial cost of the 4 MapReduce rounds dominates the execution time when the maximum length of frequent patterns is short. Moreover, Fig. 11b shows that the execution time of MG-FSM is faster than SPAMC because SPAMC spends more time for reloading frequent items. It is worth noting that when the number of frequent items increases, the difference between SPAMC-UDLT and MG-FSM becomes larger since MG-FSM spends more time scanning data in multiple MapReduce rounds, whereas SPAMC-UDLT adopts the streaming MapReduce model.

Moreover, we report the running time of writing frequent sequential patterns for Twitter dataset in terms of the percentage of total running time. Given $min_sup = 0.003$ and 0.00005%, the number of generated frequent patterns is 1,246,184–21,895,083. When maximum sequential pattern length $max_{|S|} = 4$, SPAMC-UDLT spends 28% of total running time on writing outputs. Moreover, when the maximum sequential pattern length $max_{|S|} = 12$, the percentage of total running time for writing outputs decreases to 24% since the time of

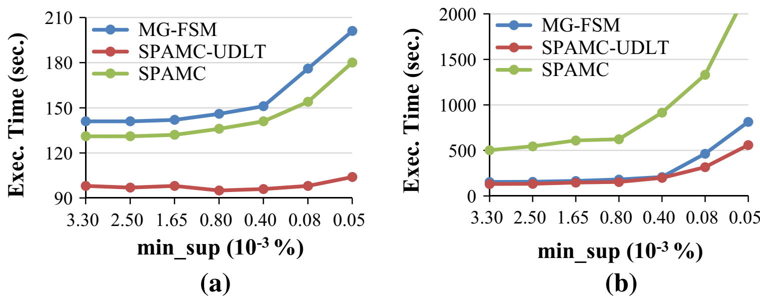


Fig. 11 Execution time with maximum length of frequent patterns ($max_{|S|}$) varied: **a** $max_{|S|} = 4$ **b** $max_{|S|} = 12$

reading candidate frequent itemsets for generating more candidate frequent itemsets is more than that of outputting the number of frequent itemsets.

5.6 Resource usage

On the basis of our experiments performed on the 12,800,000–102,400,000 transactions with distinct number of items $N = 8$ and $N = 26$. Figure 12a shows that the memory requirement of SPAMC is higher than that of SPAMC-UDLT. The dataset contains only 8 distinct items and average of 64 items per transaction; thus, the saving of memory usage in SPAMC-UDLT is not significant because the memory requirement in SPAMC, which depends on the number of distinct items, is not large. In order to measure the average memory usage in SPAMC and SPAMC-UDLT, each machine records the number of output candidate sequential patterns and the maximal number of intermediate results by a log file. Then, we summarize the log information from different machines to obtain the results of memory usage. Figure 12b shows that SPAMC-UDLT requires 40–50% of the memory size required by SPAMC, thereby reducing the memory usage of the mining phase in the streaming MapReduce model.

We analyze network traffic on the same dataset as above. Figure 13a shows the network usage of the comparison of SPAMC-UDLT and SPAMC. The bandwidth usage of SPAMC is approximately 70% that of SPAMC-UDLT, whereas the DHT needs to be distributed to all the machines. When the number of transactions is smaller with $min_sup = 0.01\%$, the DHT transmission is the major cost of the networking. Figure 13b shows that the network bandwidth requirement of SPAMC is considerably lower than that of SPAMC-UDLT, whereas SPAMC-UDLT mines frequent sequential patterns via distributed queue implementation. Thus, the frequent candidates need to be transmitted, which assists in increasing network usage. It is a

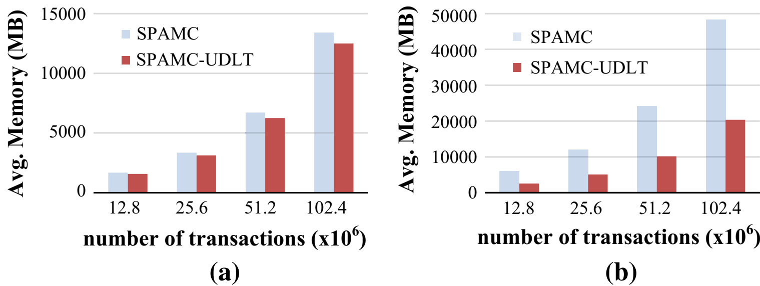


Fig. 12 Memory usage of SPAMC and SPAMC-UDLT with **a** 8 distinct items and **b** 26 distinct items

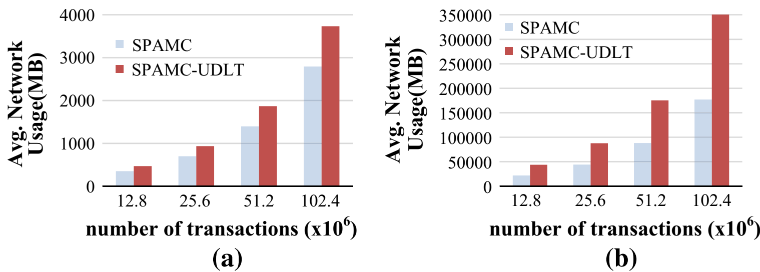


Fig. 13 Average network usage of SPAMC and SPAMC-UDLT with **a** 8 distinct items and **b** 26 distinct items

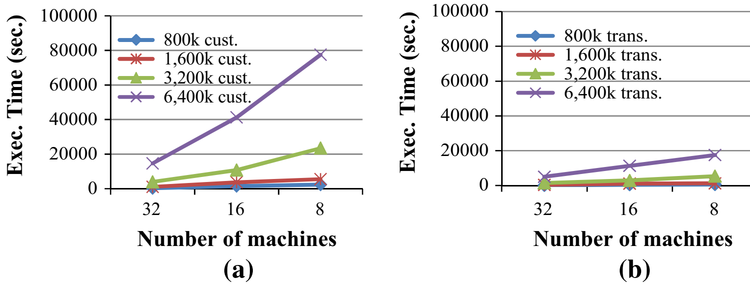


Fig. 14 Effect of the number of machines: **a** SPAMC and **b** SPAMC-UDLT

trade-off between the space usage and the network bandwidth in the proposed algorithm; in general, SPAMC-UDLT has a better performance than SPAMC for larger datasets with less min_sup for the same parameters.

5.7 Sensitivity to parameters

This section discusses the sensitivity analysis of many important parameters such as the number of machines and the number of distinct items.

5.7.1 Effect of the number of machines

The performance evaluation of various numbers of machines is conducted. We use 8, 16, and 32 machines to execute SPAMC-UDLT and SPAMC on four datasets with the number of sequences ranging from 800,000 to 6,400,000. Each sequence with 128 transactions has 64 items in average. With min_sup set to 0.01%, as shown in Fig. 14a, the execution time decreases as the number of machines increases. Note that the execution time is affected by the dataset characteristics, the computational complexity of the mining phase, and the time spent on data transmission between machines. Therefore, the time saved will not be fully proportional to the number of machines. Furthermore, as shown in Fig. 14b, SPAMC-UDLT provides nicely load balancing, removes the cost of multiround MapReduce, and thus achieves a better performance than SPAMC.

5.7.2 Effect of the number of distinct items

We report the execution time on 500,000 sequences with the number of distinct items N varying from 10 to 100. Each sequence with the average transactions per sequences $S =$ 8 and average number of items per transaction $T = 8$. Figure 15a shows the results of $min_sup = 2\%$ and $min_sup = 0.02\%$ obtained using SPAMC. Note that the execution time increases with an increase in the dataset size. This increase is mainly attributed to the facts that SPAMC-UDLT and SPAMC are based on SPAM and that SPAM takes more space to store the bitmap information when N is larger. In Fig. 15b, the curve shows that as N grows, the execution time increases. Because there are more candidate frequent patterns transmitted to the distributed queue, the execution time is determined by the UDLT access time in the network. In such a context, this experiment reveals that the advantage of SPAMC-UDLT is more prominent when the number of distinct items is small.

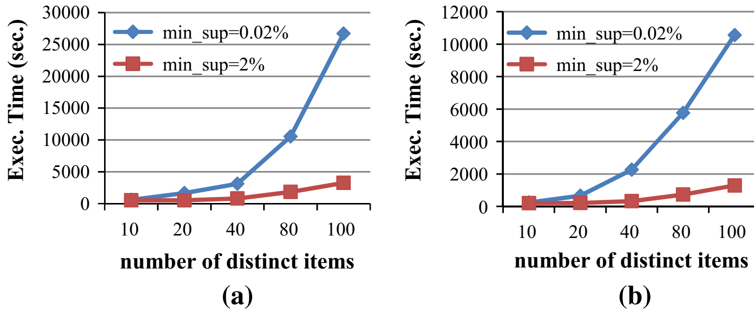


Fig. 15 Execution time with distinct items varied: **a** SPAMC and **b** SPAMC-UDLT

6 Conclusion

In this paper, we proposed SPAMC-UDLT that is a highly scalable cloud-based sequential pattern mining algorithm in streaming MapReduce model. Through streaming MapReduce model, the mining process is finished in one MapReduce job without reloading data. Besides, by applying the distributed message queue technique on UDLT, we guarantee that the mining job can be completed with limited memory and achieve nice load balancing in the cloud. Also, mappers in SPAMC-UDLT can independently execute the mining process in a distributed manner. Thus the execution time can be reduced by working on more machines very efficiently. Furthermore, the experimental results show that SPAMC-UDLT can significantly improve the scalability of sequential pattern mining in the cloud.

There are some research directions that can be investigated further. One future direction is to extend the proposed algorithm and streaming MapReduce model in a heterogeneous cloud environment. It may more effectively mine frequent sequential patterns. Also, the design concept in this paper can apply to other data mining algorithms which need to process mining procedure recursively for solving big data issues. Another direction includes the design of more effective distributed data structure for improving memory space efficiency.

References

1. Hadoop A (2012) http://hadoop.apache.org/
2. Hama A (2012) http://hama.apache.org/
3. Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proceedings of the 11th international conference on data engineering (ICDE'95), pp 3–14
4. Ayres J, Flannick J, Gehrke J et al (2002) Sequential pattern mining using a bitmap representation. In: Proceedings of the 8th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'02), pp 429–435
5. Batal I, Valizadegan H, Cooper GF et al (2013) A temporal pattern mining approach for classifying electronic health record data. Trans Intell Syst Technol (TIST'13) 63:1–22
6. Bu Y, Howe B, Balazinska M et al (2010) Haloop: efficient iterative data processing on large clusters. In: Proceedings of the VLDB endowment (PVLDB'10), pp 285–296
7. Chen CC, Tseng CY, Chen MS (2013) Highly scalable sequential pattern mining based on MapReduce model on the cloud. IEEE international congress on big data (BigData Congress'13), pp 310–317
8. Chen CC , Shuai HH, and Chen MS (2016) Appendix of distributed and scalable sequential pattern mining through stream processing. https://www.csie.ntu.edu.tw/~d96944011/kais2016/appendix
9. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM (CACM'08) 51:107–113

10. Ekanayake J, Li H, Zhang B et al (2010) Twister: a runtime for iterative MapReduce. In: Proceeding of the 19th ACM international symposium on high performance distributed computing (HPDC'10), pp 810–818
11. Fang W, Lu M, Xiao X et al (2009) Frequent itemset mining on graphics processors. In: Proceedings of the 5th international workshop on data management on new hardware (DaMoN'09), pp 34–42
12. Gomariz A, Campos M, Marin R et al (2013) ClaSP: an efficient algorithm for mining frequent closed sequences. In: Proceedings of the 17th Pacific-Asia conference on knowledge discovery and data mining (PAKDD'13), pp 50–61
13. Goodhope K, Koshy J, Kreps J et al (2012) Building LinkedIn's real-time activity data pipeline. IEEE Data Eng Bull (Data Eng Bull'12) 35:33–45
14. Guralnik V, Karypis G (2004) Parallel tree-projection-based sequence mining algorithms. Parallel Comput (PARALLEL COMPUT'04) 30:443–472
15. Han J, Pei J, Mortazavi-Asl B et al (2000) FreeSpan: frequent pattern-projected sequential pattern mining. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'00), pp 355–359
16. Han J, Pei J, Yan X (2005) Sequential pattern mining by pattern-growth: principles and extension. Foundations and advances in data mining. Springer, Berlin
17. Ho J, Lukov L, Chawla S (2005) Sequential pattern mining with constraints on large protein databases. In: Proceedings of the 12th international conference on management of data (COMAD'05), pp 89–100
18. Huang JW, Tseng CY, Ou JC et al (2008) A general model for sequential pattern mining with a progressive database. IEEE Trans Knowl Data Eng (TKDE'08) 20:1153–1167
19. Huang JW, Lin SC, Chen MS (2010) DPSP: distributed progressive sequential pattern mining on the cloud. 14th Pacific–Asia conference on knowledge discovery and data mining (PAKDD'10), pp 27–34
20. Isard M, Budiu M, Yu Y et al (2007) Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Oper Syst Rev (SIGOPS'07) 41:59–72
21. Ji X, Bailey J, Dong G (2007) Mining minimal distinguishing subsequence patterns with gap constraints. Knowl Inf Syst (KAIS'07) 11:259–286
22. Kreps J, Narkhede N, Rao J (2011) Kafka: a distributed messaging system for log processing. NetDB workshop
23. Liao CC, Chen MS (2014) DFSP: a Depth-First SPelling algorithm for sequential pattern mining of biological sequences. Knowl Inf Syst (KAIS'14) 38:623–639
24. Luo C, Chung S (2008) A scalable algorithm for mining maximal frequent sequences using a sample. Knowl Inf Syst (KAIS'08) 15:149–179
25. Mabroukeh NR, Ezeife CI (2010) A taxonomy of sequential pattern mining algorithms. ACM Comput Surv (CSUR'10) 43:1–41
26. Mane RV (2013) A comparative study of Spam and PrefixSpan sequential pattern mining algorithm for protein sequences. In: Proceedings of the 3rd international conference on advances in computing, communication, and control (ICAC3'13), pp 147–155
27. Miliaraki I, Berberich K, Gemulla R et al (2013) Mind the gap: large-scale frequent sequence mining. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data (SIGMOD'13), pp 797–808
28. Papapetrou P, Kollios G, Sclaroff S et al (2009) Mining frequent arrangements of temporal intervals. Knowl Inf Syst (KAIS'09) 21:133–171
29. Parimala M, Sathiyabama S (2012) SPMLS: an efficient sequential pattern mining algorithm with candidate generation and frequency testing. Int J Comput Sci Eng (IJCSE'12) 4:601–607
30. Pei J, Han J, Mortazavi-asl B et al (2001) PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In: Proceedings of the 7th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'01), pp 215–224
31. Perer A, Wang F (2014) Frequence: interactive mining and visualization of temporal frequent event sequences. In: Proceedings of the 19th ACM international conference on intelligent user interfaces (IUI'14), pp 153–162
32. Sahli M, Mansour E, Kalnis P (2014) ACME: a scalable parallel system for extracting frequent patterns from a very long sequence. VLDB J (VLDBJ'14) 23:871–893
33. Shie BE, Hsiao HF, Tseng V (2013) Efficient algorithms for discovering high utility user behavior patterns in mobile commerce environments. Knowl Inf Syst (KAIS'13) 37:363–387
34. Srikant R, Agrawal R (1996) Mining sequential patterns: generalizations and performance improvements. In: Proceedings of the 5th international conference on extending database technology (EDBT'96), pp 3–17
35. Samza (2013) https://samza.incubator.apache.org/
36. Storm: distributed and fault–tolerant realtime computation (2012) http://storm.incubator.apache.org/
37. Spark: Lightning-fast cluster computing (2013) https://spark.incubator.apache.org/

38. S4: Distributed Stream Computing Platform (2010) https://incubator.apache.org/s4/
39. Twister: iterative MapReduce (2012) https://iterativemapreduce.org/
40. White Tom (2009) Hadoop: the definitive guide. O'Reilly Media, Newton
41. Wang K, Xu Y, Yu JX (2004) Scalable sequential pattern mining for biological sequences. In: Proceedings of the 13th ACM international conference on information and knowledge management (CIKM'04), pp 178–187
42. Wang X, Wang J, Wang T et al (2010) Parallel sequential pattern mining by transaction decomposition. International conference on fuzzy systems and knowledge discovery (FSKD'10), pp 1746–1750
43. Weng L, Menczer F, Ahn YY (2013) Virality prediction and community structure in social networks. Sci Rep 3. doi:10.1038/srep02522
44. Wu K, Otoo EJ, Shoshani A (2002) Compressing bitmap indexes for faster search operations. In: Proceedings of 14th international conference on scientific and statistical database management (SSDBM'02), pp 99–108
45. Yu D, Wu W, Zheng S et al (2012) BIDE-based parallel mining of frequent closed sequences with MapReduce. In: Proceedings of the 12th international conference on algorithms and architectures for parallel processing (ICA3PP'12), pp 177–186
46. Yu D, Zhu Q, Shao J et al (2014) Parallel execution of data-intensive web services based on data-flow constructs and I/O operation ratio. Int J Database Theory Appl (IJDTA'14) 7:129–138
47. Zaharia M, Chowdhury M, Das T et al (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on networked systems design and implementation (NSDI'12), p 2
48. Zaharia M, Chowdhury M, Das T et al (2012) Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: Proceedings of the 4th USENIX conference on hot topics in cloud computing (HotCloud'12), pp 215–224
49. Zaki MJ (1998) Efficient enumeration of frequent sequences. In: Proceedings of the 7th ACM international conference on information and knowledge management (CIKM'98), pp 68–75
50. Zaki MJ (2001) Parallel sequence mining on shared-memory machines. J Parallel Distrib Comput (JPDC'01) 61:401–426
51. Zhao Q, Bhowmick SS (2003) Sequential pattern matching: a survey. ITechnical report CAIS Nayang Technological University Singapore, pp 1–26

Chun-Chieh Chen received the M.S. degree from the Department of Computer Science and Information Engineering, National Central University, Taiwan, in 2006, and the Ph.D. degree at the Graduate Institute of Networking and Multimedia at the National Taiwan University, Taipei, Taiwan, in 2016. From 2009 to 2016, he is a member of Network Database Lab (NetDB Lab). He was an member of Phi Tau Phi Scholastic Honor Society in 2006 and 2016. His research interests include multimedia applications, big data, data mining, and cloud computing.

Hong-Han Shuai received the B.S. degree from the Department of Electrical Engineering, National Taiwan University (NTU), Taipei, Taiwan, R.O.C., in 2007, the M.S. degree in computer science from NTU in 2009, and the Ph.D. degree in Graduate Institute of Communication Engineering in 2015. He is now an assistant professor in NCTU. His research interests are in the area of social network analysis, data mining, and video processing.

Ming-Syan Chen received the Ph.D. degrees in Computer, Information and Control Engineering from The University of Michigan, Ann Arbor, MI, USA. He is now the Dean of the College of Electrical Engineering and Computer Science and also a Distinguished Professor in EE Department at National Taiwan University. He was a research staff member at IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA from 1988 to 1996, the Director of GICE from 2003 to 2006, the President/CEO of Institute for Information Industry (III), which is one of the largest organizations for information technology in Taiwan, from 2007 to 2008, and also a Distinguished Research Fellow and the Director of Research Center of Information Technology Innovation (CITI) in the Academia Sinica from 2008 to 2015. His research interests include databases, data mining, social networks, and multimedia networking, and he has published more than 350 papers in his research areas. Dr. Chen is a Fellow of ACM and a Fellow of IEEE.