World Wide Web (2019) 22:295–324
https://doi.org/10.1007/s11280-018-0566-1

Scalable and parallel sequential pattern mining using spark

Xiao Yu [1,2] · Qing Li [2] · Jin Liu [1]

Received: 14 November 2017 / Revised: 20 February 2018 / Accepted: 5 April 2018 /
Published online: 10 May 2018

Abstract The performance of the existing parallel sequential pattern mining algorithms is often unsatisfactory due to high IO overhead and imbalanced load among the computing nodes. To address such problems, this paper proposes two efficient parallel sequential pattern mining algorithms based on Spark, i.e., GSP-S (GSP algorithm based on Spark) and PrefixSpan-S (PrefixSpan algorithm based on Spark). For both algorithms, multiple MapReduce jobs are implemented to complete a mining task. To reduce IO overhead and take advantage of cluster memory, the first MapReduce job loads sequence database from the Hadoop Distributed File System (HDFS) into the Spark resilient distributed datasets (RDDs), and further MapReduce jobs read the database from the RDDs and store intermediate results back into the RDDs. Our findings suggest that a wise choice can be made between GSP-S and PrefixSpan-S, depending on the user-specified minimum support threshold. Moreover, theoretical analysis shows that GSP-S and PrefixSpan-S are sensitive to data distribution on the cluster. To further improve performance, we propose two database partition strategies to balance load among the computing nodes in a cluster. Experiment results demonstrate the high performance of GSP-S and PrefixSpan-S in terms of load-balancing, speedup and scalability.

Keywords Sequential pattern mining · MapReduce · Load balance · Spark

✉ Jin Liu
jinliu@whu.edu.cn

Xiao Yu
xiaoyu_whu@yahoo.com

[1] State Key Lab. of Software Engineering, School of Computer Science, Wuhan University, Wuhan 430072, China

[2] Department of Computer Science, City University of Hong Kong, Hong Kong 999077, China

1 Introduction

1.1 Motivation

With the exponential growth of data and complexity of intelligent systems, fast machine learning and computational intelligence techniques are needed. Sequential pattern mining is an essential machine learning technique used in many domains, such as the discovery of frequent trajectory patterns [33] and user behavior patterns [47], the analysis of DNA sequence [19], and the study of customer purchase behaviors [7]. The need for efficient and fast implementation of sequential pattern mining algorithms for handling massive data poses some research challenges in these domains. In particular, traditional sequential pattern mining algorithms, like GSP algorithm [29], PrefixSpan algorithm [23, 24], and others [2, 4, 14, 44], face bottlenecks in dealing with large-scale datasets. Therefore, many such algorithms are parallelized based on some distributed frameworks better suited for handling massive datasets. In recent years, Spark [42] has been a popular distributed computing platform, which is a standard open source implementation for the MapReduce programming framework [9].

Converting a serial sequential pattern mining algorithm into a parallel algorithm based on Spark may sound non-challenging, but the following important factors must be considered in order to achieve the high performance of parallel sequential pattern mining algorithms based on Spark:

1) *IO overhead*. The multiple-database-scans feature of GSP necessitates multiple MapReduce jobs. PrefixSpan's recursive mining of projected databases also brings with it the demand for multiple MapReduce jobs. The i-th MapReduce job needs to read the output from the (i-1)-th MapReduce job as the input. Each MapReduce job also needs to read sequence databases or projected databases. If the output of each MapReduce job and databases are stored in the HDFS, it leads to high IO overhead. Therefore, reducing the IO overhead becomes a critical aspect of algorithmic design.
2) *Load balancing*. In a Spark cluster, the master has to set the default configuration to initialize each MapReduce job. Reducers cannot start until all mappers have finished the assigned computing tasks. Imbalanced load forces reducers to wait for the slowest mapper, thereby leading to pure waiting overhead. In reality, sequences in the sequence database are often of different length and highly skewed. It is therefore important to study how to split the database and distribute the computing load to different nodes in the cluster so as to achieve good load-balancing performance.

1.2 Our work and contributions

Accordingly, we propose two efficient parallel sequential pattern mining algorithms based on Spark, i.e., GSP-S (GSP algorithm based on Spark) and PrefixSpan-S (PrefixSpan algorithm based on Spark). Compared with the existing parallel sequential pattern mining algorithms, GSP-S and PrefixSpan-S have distinctive features, i.e., low IO overhead and balanced load. From the IO overhead perspective, for both GSP-S and PrefixSpan-S, multiple MapReduce jobs are implemented to complete the mining task. To reduce IO overhead and take advantage of cluster memory, the first MapReduce job loads the sequence database from the HDFS into the Spark RDDs, and subsequent MapReduce jobs read the database from the RDDs and store intermediate results back into the RDDs. Moreover, to tackle the problem of imbalanced load among computing nodes, our theoretical analysis has shown that GSP-S and PrefixSpan-S are sensitive to data distribution, because GSP-S needs multiple database scans to identify the candidate sequence, and PrefixSpan-S needs to scan the previous projected databases to construct new projected databases. To improve the algorithms' performance, we propose two database partition strategies to well balance load among the computing nodes in the cluster. Experiment results demonstrate that GSP-S and PrefixSpan-S can reduce IO overhead and balance load to achieve significant performance in terms of load-balancing, speedup and scalability on Spark clusters.

The main contributions of our paper can be summarized as follows.

(1) We propose two efficient parallel sequential pattern mining algorithms based on Spark, i.e., GSP-S and PrefixSpan-S, which have low IO overhead by adopting in-memory computation. The IO overhead of GSP-S and PrefixSpan-S are only $O(|D| \times l + \sum_{k \geq 1} (|L_k| \times k))$, where $|D|$ is the number of sequences in the database D, l is the average length of the sequences, and $|L_k|$ is the number of all k-sequential patterns.
(2) We conduct a theoretical investigation on the factors that incur the imbalanced load of GSP-S and PrefixSpan-S, and propose two database partition strategies to balance load well among computing nodes in Spark cluster. Compared with using the Spark default settings to split the original sequence database, our database partition strategies can reduce the runtime of GSP-S at least by 18.97%, and reduce the runtime of PrefixSpan-S at least by 21.85%.
(3) We provide a thorough analytical study of the time complexity, IO overhead and network overhead of the proposed algorithms, and theoretically prove that the time complexity of GSP-S and PrefixSpan-S is reduced by n times approximately compared with the serial algorithms, where n is the number of database partitions.

1.3 Organization

The remainder of this paper is organized as follows. Section 2 examines earlier works related to our problem. Section 3 describes some background knowledge. Section 4 proposes the GSP-S algorithm and describes the implementation details. Section 5 proposes the PrefixSpan-S algorithm and provides the implementation details. Section 6 demonstrates the experimental results and shows the comparisons with the state-of-the-art approaches. Finally, we conclude this paper in Section 7.

2 Related work

In this section, we first review the related work on serial sequential pattern mining algorithms, and then focus on parallel algorithms under distributed computing platforms.

2.1 Sequential pattern mining

Serial sequential pattern mining algorithms can be divided into three camps, namely, Apriori-based algorithms [2, 29] and pattern growth algorithms [14, 23, 24] and vertical format-based algorithms [4, 44].

The AprioriAll [2] algorithm sets the basic for a variety of Apriori-based algorithms. In the following year, the GSP algorithm [29] was proposed, and it outperforms AprioriAll because of the more intelligent candidate sequence generation method. However, the main bottleneck of these algorithms is that they need to scan the sequence database repeatedly to discover all sequential patterns, thus leading to high time cost.

To improve the performance of Apriori-based algorithms, Han et al. proposed two projected database-based algorithms, called FreeSpan [14] and PrefixSpan [23, 24], as a way to avoid scanning the sequence database repeatedly. These algorithms construct the projected database, and then recursively mine them to discover sequential patterns. FreeSpan and PrefixSpan utilize the construction of projected databases and do not require multiple database scans to discover the sequential pattern. But if the minimum support is low, they need to generate large number of projected databases, and the cost is nontrivial [23].

The vertical format-based algorithms transform the original database into vertical format to quickly find all sequential patterns. The SPAM algorithm [4] first transforms the entire sequence database into a vertical bitmap representation, and then uses a depth-first traversal strategy to discover all sequential patterns. The SPADE algorithm [44] discovers all sequential patterns in only three database scans by using lattice search techniques based on a vertical database format. However, the vertical format-based algorithms need completely fit the sequence database into the main memory and require more memory space.

In addition, there are various kinds of extensions for sequential pattern mining, including multi-dimensional sequential pattern mining [25, 39], maximal sequential pattern mining [10, 11, 21, 22] and closed sequential pattern mining [18, 31, 32, 38].

2.2 Parallel mining of sequential patterns

With the exponential growth of data and complexity of systems, the above-mentioned serial algorithms face bottlenecks in dealing with massive datasets. To solve the problem, some parallel algorithms have been proposed for handling massive datasets.

Shintani et al. [27] proposed three parallel algorithms (NPSPM, SPSPM and HPSPM) for mining sequential patterns on a shared- nothing environment. Among three algorithms, HPSPM attains best performance. Zaki [45] extended his serial sequential pattern mining algorithm (SPADE) to the shared memory parallel architecture, creating pSPADE. Experimental results showed that pSPADE can achieve good speedup and excellent scaleup. Gurainik et al. [12] presented two parallel sequential pattern mining algorithms based on the distributed memory system. Cong et al. [8] proposed the Par-CSP algorithm on the distributed memory system to mine closed sequential patterns. Zhang et al. [46] proposed the FMGSP algorithm to mine global sequential patterns on the distributed system. Experimental results indicated that the performance of FMGSP was predominant for large databases. Wu et al. [35] proposed parallel GSP algorithm based on the grid computing platform. Kessl et al. [16] proposed a parallel PrefixSpan algorithm using static load-balancing on the distributed system. However, these algorithms are without fault tolerance, because they are mainly implemented on the shared memory system, the distributed memory system or the grid computing platform that provide little support for fault tolerance.

The MapReduce programming framework [9] and its implementation in Hadoop [13] offer us an ideal environment for the implementation of parallel algorithms, because of the fault-tolerant mechanism and the ease of use. These Hadoop-based algorithms can be mainly divided into two camps, namely, iterative algorithms and non-iterative algorithms. Huang et al. [15] proposed the DPSP algorithm based on Hadoop for progressive sequential pattern mining. Chen et al. [6] proposed the SPAMC algorithm based on Hadoop for sequential pattern mining. Yu et al. [40] proposed the BIDE-MR algorithm based on Hadoop for closed sequential pattern mining. Wei et al. [34] and Puspita et al. [26] proposed parallel PrefixSpan algorithm based on Hadoop to mine large-scale datasets. These iterative algorithms adopt multiple MapReduce jobs to implement parallel sequential pattern mining on Hadoop. Each job needs to perform a read-write operation to the HDFS, which leads to high IO overhead and time cost. In addition, these algorithms do not take load balance into consideration, which is quite important for mining massive datasets.

Moreover, DGSP [41] and PTDS [30] are non-iterative parallel algorithms based on Hadoop, derived from GSP algorithm and PrefixSpan algorithm respectively. They first decompose the sequence database, and then apply serial GSP algorithm or PrefixSpan algorithm on the set of subsequences to generate local sequential patterns, finally combine the mining results together. However, these non-iterative algorithms cannot efficiently maintain load balance, because it is difficult to ensure that each computing node is assigned with the same amount of computing load. In addition, these algorithms generate redundant local sequential patterns in the set of subsequences, thus leading to extra time cost.

In this paper, we develop two parallel sequential pattern mining algorithms based on Spark. i.e., GSP-S (GSP algorithm based on Spark) and PrefixSpan-S (PrefixSpan algorithm based on Spark). The reasons we parallelize GSP and PrefixSpan are as follows. GSP and PrefixSpan are the most representative and classical Apriori-based and pattern growth approaches, respectively. Most of extended sequential pattern mining algorithms (e.g., multi-dimensional sequential pattern mining algorithms, maximal sequential pattern mining algorithms and closed sequential pattern mining algorithms) are Apriori-based or pattern growth approaches instead of vertical format-based [1]. For example, Pinto et al. [25] proposed Seq-Dim and Dim-Seq algorithms to mine multi-dimensional sequential patterns, which divide the mining process into two steps. Seq-Dim algorithm first mines sequential patterns, and then for each sequential pattern, forms projected multi-dimensional database and finds multi-dimensional patterns within the projected databases, while Dim-Seq algorithm uses the reverse procedure. Yu and Chen [39] introduced two algorithms, the first of which is developed by modifying the traditional Apriori algorithm and the second by modifying the PrefixSpan algorithm. The first algorithm has different methods for candidate generation and support counting compared with the original Apriori algorithm. The second algorithm has different approaches for sequential pattern growth and projected database construction compared with the original PrefixSpan algorithm. For both algorithms, different dimensional scopes of each element are considered as the key factor for algorithm design. AprioriAdjust [21] is an Apriori-based algorithm for mining maximal sequential pattern, while MaxSP [10] is a pattern growth based algorithm inspired by PrefixSpan. For developers which desire to develop parallel extended sequential pattern mining algorithms, they can refer to the parallel frameworks of GSP-S and PrefixSpan-S without paying too much efforts in re-designing new parallel architectures.

3 Preliminary

In this section, we first review the definition of sequential pattern mining. Then, we briefly introduce the MapReduce programming framework and Spark platform.

3.1 Sequential pattern mining

Definition 1 Let $I = \{item_1, item_2, \ldots, item_n\}$ be a set of n different items, which comprise the alphabet. An itemset is a subset of items and denoted by ($item_1\ item_2 \ldots item_l$), where $item_k$ is an item. It is assumed that items in an itemset are sorted in lexicographic order and can occur at most once in an itemset. A sequence $s = \langle s_1, s_2, \ldots, s_m \rangle$ is an ordered list of itemsets, where s_j is an itemset.

Definition 2 A sequence database D is a set of tuples (sid,s), where sid is a sequence-id, and s is a sequence. A sequence database is shown in Table 1.

Definition 3 A sequence $\beta = \langle b_1, b_2, \ldots, b_m \rangle$ is called a subsequence of another sequence $\alpha = \langle a_1, a_2, .., a_n \rangle$ if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n \leq n$ such that $b_1 \subseteq a_{j1}, b_2 \subseteq a_{j2}, \ldots, b_n \subseteq a_{jn}$. We also call α contains β.

Definition 4 The support count of a sequence s in D is defined as the number of sequences in D containing s, denoted as $sup_D(s)$.

Definition 5 Given a minimum support threshold $minsup$, a sequence s is called a (frequent) sequential pattern in sequence database D if $sup_D(s) \geq minsup$. A sequential pattern containing k items is called a k-sequential pattern. The set of k-sequential patterns is defined as L_k.

Definition 6 Given a sequence $\beta = \langle b_1, b_2, \ldots, b_n \rangle$, a sequence $\alpha = \langle a_1, a_2, .., a_m \rangle$ $(m \leq n)$ is called a prefix of β if and only if $b_i = a_i$ ($i \leq m$-1,$a_m \subset b_m$), and either b_m-$a_m = \Phi$ or all the frequent items in (b_m-a_m) are larger than those in a_m according to the total ordering. The sequence $\gamma = \langle b_m$-$a_m, b_{m+1}, \ldots, b_n \rangle$ is called the suffix β of relative to prefixα.

For example, $\langle a \rangle$, $\langle a\ c \rangle$ and $\langle a\ c\ g \rangle$ are prefixs of sequence $\langle a\ c\ g\ h \rangle$. $\langle c\ g\ h \rangle$ is the suffix relative to prefix $\langle a \rangle$, $\langle g\ h \rangle$ is the suffix relative to prefix $\langle a\ c \rangle$, and $\langle h \rangle$ is the suffix relative to prefix $\langle a\ c\ g \rangle$.

Definition 7 Let α be a sequential pattern in D. The α-projected database is the collection of suffixes of sequences in D with regard to prefix α, denoted as $S|_\alpha$.

Given an input sequence database D and a minimum support threshold $minsup$, sequential pattern mining is to find all sequential patterns whose support is not less than $minsup$.

Table 1 A sequence database

sid	*S*
S_1	<*a c g h*>
S_2	<(*c d*) (*e f g*)>
S_3	<*h*>
S_4	<*c g*>
S_5	<*g a*>
S_6	<(*a b*) *a c*>

Example Table 1 shows a database with 6 sequences. With *minsup* = 2, Table 2 shows 6 sequential patterns as well as the corresponding projected databases.

3.2 MapReduce and spark

MapReduce [9] is a programming framework that abstracts computation problems through two functions: *map* and *reduce*. A computing node in MapReduce is called a mapper or a reducer. A mapper takes in the input key/value pairs and applies the map function to generate a set of intermediate key/value pairs. A reducer aggregates all the values with the same key and applies the reduce function to the values [20, 36].

Apache Hadoop [13] was established as the standard open source implementation for the MapReduce programming framework. However, it has been quickly succeeded by Apache Spark [42], which proposed an improved data abstraction called the RDDs [43] to support in-memory computation. Spark runs on top of existing Hadoop cluster and accesses the HDFS. RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage). If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

4 Parallelization of GSP

In this section, we first review the standard GSP algorithm [29] and then propose our GSP-S algorithm. We also propose a sequence database partition strategy to balance load among the computing nodes in a Spark cluster. Finally, the time complexity analysis, IO overhead and network overhead analysis of GSP-S are conducted.

4.1 GSP algorithm

The GSP algorithm makes multiple database scans to find all sequential patterns. The first scan computes the support of each item to find L_1. From L_1, the candidate *2*-sequences (C_2) are generated, and another database scan computes their support count to find L_2. This process is repeated until no more sequential patterns are found or no candidate sequences are generated. There are two main steps in the candidate generation:

Table 2 Sequential pattern

sequential pattern (prefix)	support count	projected database
<*a*>	3	<*c g h*>, <*a c*>
<*c*>	4	<*g*>, <*g*>, <*g h*>
<*g*>	4	<*a*>
<*h*>	2	*null*
<*c g*>	3	*null*
<*a c*>	2	<*g h*>

Join Phase- C_k are generated by joining L_{k-1} with L_{k-1}. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2. The candidate sequence generated by joining s_1 with s_2 is the sequence s_1 extended with the last item in s_2.

Prune Phase- Candidate sequences that have a (k-1)-subsequence whose support is less than the minimum support are deleted.

The pseudocode of GSP is given below as Algorithm 1.

Algorithm 1 GSP algorithm

```
Input: sequence database ($D$), $minsup$
Output: all sequential pattern ($L$)
1:$L_1 = \{1\text{-sequential pattern}\}$;
2:for ($k$=2; $L_{k-1} \neq \varphi$; $k$++) do
3:    $C_k = getCandidate(L_{k-1})$;
4:    for all sequence $s$ in $D$ do
5:        increment the count of all candidates $c$ in $C_k$ that are contained in $s$;
6:    end for
7:    $L_k = \{c \in C_k \mid c.\text{sup\_count} \geq minsup\}$;
8: end for
9: $L = \cup_k L_k$;
```

The time complexity of GSP is $O\left(|D| \times l + \sum_{k \geq 2} (|L_k| \times |L_k| + |C_k| + |D| \times |C_k|)\right)$, where $|D|$ is the number of sequences in D, l is the average length of the sequences, $|L_k|$ is the number of sequential patterns in L_k, and $|C_k|$ is the number of sequence in C_k.

Proof Line 1 requires $O(|D| \times l)$ time for scanning the database to find L_1. In further database scan, line 3 requires $O(|L_k| \times |L_k| + |C_k|)$ time to generate candidate sequences and lines 4–6 require $O(|D| \times |C_k|)$ time to compute the count of the candidate sequences. Thus, the time complexity of GSP is $O(\text{GSP}) = O\left(|D| \times l + \sum_{k \geq 2} (|L_k| \times |L_k| + |C_k| + |D| \times |C_k|)\right)$.

The preceding analysis shows that GSP bears several disadvantages, such as generating a huge number of candidate sequences and performing multiple database scans. The multiple-database-scans feature of GSP necessitates multiple MapReduce jobs. However, the implement of RDDs makes Spark especially suitable and useful for parallel processing of distributed data with iterative algorithms [43]. In addition, GSP is level-wise in nature, and each stage in GSP is loosely coupled [35]. Each stage in GSP accepts the outputs from its previous stage and produces inputs for its next stage. Once launched with proper inputs, a stage goes individually without interfering the previous one [35]. It is more convenient to parallel the GSP algorithm based on Spark without considerable attention to re-design new parallel architecture. In the following, we introduce our proposed GSP-S algorithm.

4.2 GSP-S

GSP-S consists of three important phases: database partition phase, preparation phase and mining phase. Figure 1 shows the framework of GSP-S. In the database partition phase, GSP-S splits a sequence database into n database partitions based on our proposed database partition strategy. The details of the database partition phase will be discussed in Section 4.3. The

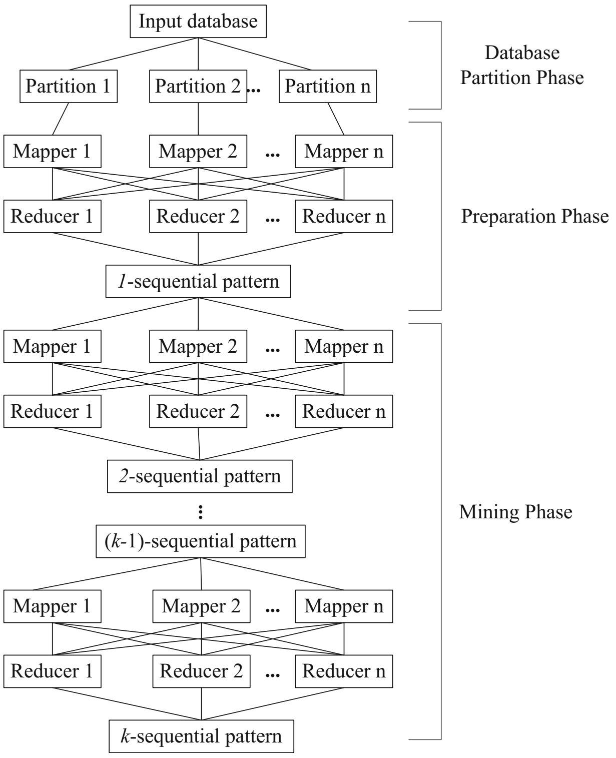


Figure 1 Framework of GSP-S algorithm

preparation phase and mining phase perform the actual mining task to generate all sequential patterns through iterative MapReduce jobs. The mappers of *i*-th job generate the candidate *i*-sequences (C_i) and compute the support count of the candidate sequences in each database partition. The reducers of *i*-th job then obtain the final sequential pattern by aggregating their local support count.

To reduce IO overhead and take advantage of cluster memory, the first MapReduce job loads the sequence database from the HDFS into the Spark RDDs, and subsequent MapReduce jobs read the sequence database from the RDDs and store sequential patterns generated in each MapReduce job into the RDDs. Figure 2 shows the lineage graph for the RDDs in GSP-S. We present the details of the preparation phase and the mining phase immediately below.

Preparation Phase In this phase, we use a MapReduce job to generate all 1-sequential patterns (L_1). Database partitions generated in the database partition phase are loaded into the Spark RDDs from the HDFS, so as to reduce IO overhead and take the advantage of the cluster memory. As illustrated in the left-hand side of Figure 2, each mapper invokes the first *flatMap*() function to read sequences in database partitions, where each sequence is stored in the format of <LongWritable offset, Text sequence> key/value pair, and then invokes the other *flatMap*() function to split the sequence into items. Next, each mapper applies *map*() function to yield the <item, 1 > key/value pairs. Note that identical items in a row of sequence are counted as only one occurrence. These key/value pairs with the same key are merged in a specific reducer. Finally, each reducer invokes the *reducebyKey*() function to aggregate the

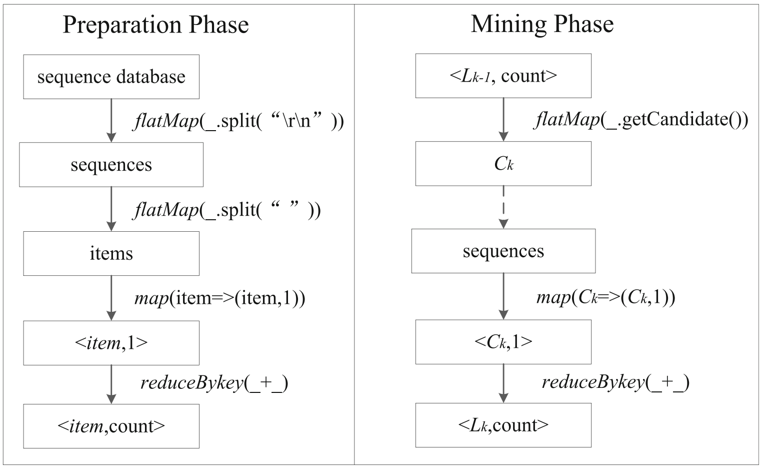


Figure 2 Lineage graph for the RDDs in GSP-S

support count of the items, and output <item, *sum* > as 1-sequential patterns (L_1) when the *sum* is not less than *minsup*. The pseudocode of the preparation phase is detailed in Algorithm 2.

Algorithm 2 Preparation phase of GSP-S algorithm

```
Input: database partitions ($D_i$), minsup
Output: 1-sequential pattern ($L_1$)
1:  for all sequence s in $D_i$ do
2:      flatMap(offset,s);
3:      for all item i in s do
4:          output <item,1>;
5:      end for
6:  end for
7:  reduceBykey(key=i, value=1);
8:  sum=0;
9:  for all item i do
10:     for all value v in item i's value list do
11:         sum+=v;
12:     end for
13: end for
14: if $sum \geq minsup$ then
15:     output<i,sum>;
16: end if
```

Example Figure 3 illustrates an execution example of the preparation phase. Three database partitions generated in the database partition phase are assigned to three mappers. The first mapper handles sequences <(*c d*) (*e f g*) > and < *h* > and outputs <*c*,1>, <*d*,1>, <*e*,1>, <*f*,1>, <*g*,1 > and < *h*,1 > key/value pairs. The other two mappers handle assigned database partitions and output key/value pairs in the same manner. The reducer aggregates the support count and outputs 1-sequential patterns <*a*,3>, <*c*,4>, <*g*,4 > and < *h*,2 > .

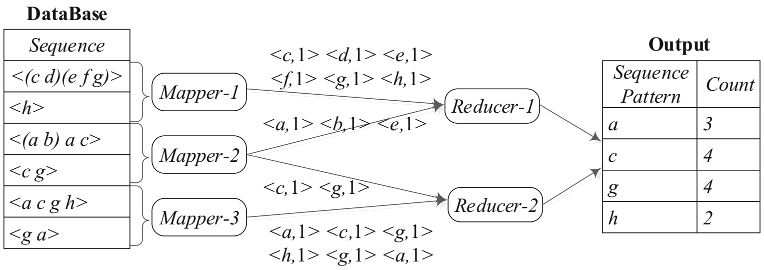


Figure 3 An execution example of the preparation phase

Mining Phase This phase discovers all sequential patterns through iterative MapReduce jobs. The preparation phase generates 1-sequential patterns and stores them in the RDDs rather than the HDFS to reduce IO overhead. As illustrated in the right-hand side of Figure 2, at *k*-th MapReduce job, each mapper reads L_{k-1} from the RDDs to generate candidate *k*-sequences (C_k) via the candidate sequence generation procedure. Then, a *map*() function is applied to read each sequence *s* in the database partition from the RDDs, use the *subsequence()* function to identify all candidates in C_k, and yield the <*c*,1 > key/value pairs for the candidate *c* that is contained in *s*. These key/value pairs with the same key are merged in a specific reducer. Finally, each reducer invokes a *reduceBykey*() function to aggregate the support count of the candidate sequences, and output the <*c*, *sum* > key/value pairs as *k*-sequential patterns (L_k) when the *sum* is not less than *minsup*. The pseudocode of the mining phase is detailed in Algorithm 3.

Algorithm 3 Mining phase of GSP-S algorithm

```
Input: database partitions ($D_i$),minsup, 1-sequential pattern ($L_1$)
Output: k-sequential pattern ($L_k$)
1:  Read $L_{k-1}$ from the RDDs;
2:  $C_k$= getCandidate ($L_{k-1}$);
3:  for all sequence s in $D_i$ do
4:      flatMap(offset,s);
5:      for all candidate c in $C_{k+1}$
6:          if s.subsequence (c) then
7:              output<c,1> ;
8:          end if
9:      end for
10: end for
11: reduceBykey(key= c, value=1);
12: sum=0;
13: for all  candidate c  do
14:     for all value v in candidate c's value list do
15:         sum+=v;
16:     end for
17: end for
18: if sum ≥ minsup then
19:     output<c,sum>;
20: end if
```

Example Figure 4 illustrates an execution example of the first MapReduce job of the mining phase. Each mapper reads L_1 from the RDDs to generate C_2. The first mapper reads sequence <(*c d*) (*e f g*)> from the RDDs, finds the sequence containing the candidate 2-sequence <*c g*>, and outputs <*c g*, 1> key/value pair. Then, the mapper reads sequence <*h*> from the RDDs, finds the sequence not containing any candidate 2-sequence and outputs nothing. The other two mappers read the assigned database partitions from the RDDs and output key/value pairs in the same manner. The reducer aggregates the support count of the candidate 2-sequences and outputs 2-sequential patterns <*c g*, 3> and <*a c*, 2>.

4.3 Load balance

In order to balance load among the computing nodes of a Spark cluster, we first need to quantitatively measure the total computing load of each computing node. The computing cost of GSP-S mostly occurs in scanning the database partition to identity the candidate sequences, so the imbalance load problem of each computing node is mainly induced by the *subsequence()* function which has a significant impact on GSP-S. Therefore, we pay particular attention to the *subsequence()* function in the mining phase.

The *subsequence()* function of the mining phase identifies whether a candidate sequence c is contained in s or not. Therefore, the time complexity of *subsequence()* function is $O(l \times m)$, where m is the length of c and l is the length of s. We consider a sequence database split into n partitions across n computing nodes. As the i-th computing node needs to scan the assigned i-th database partition D_i to identify all candidate sequences in C_k, the computing load of the i-th computing node is $O(|D_\mathrm{i}| \times |C_k| \times l \times m)$. Since the values of $|C_k|$ and m are fixed, the computing load of each computing node is proportional to the total length of sequences in D_i. However, in real life, sequences in a sequence database are often of different length and highly skewed. Therefore, care should be taken during the process of data partitioning.

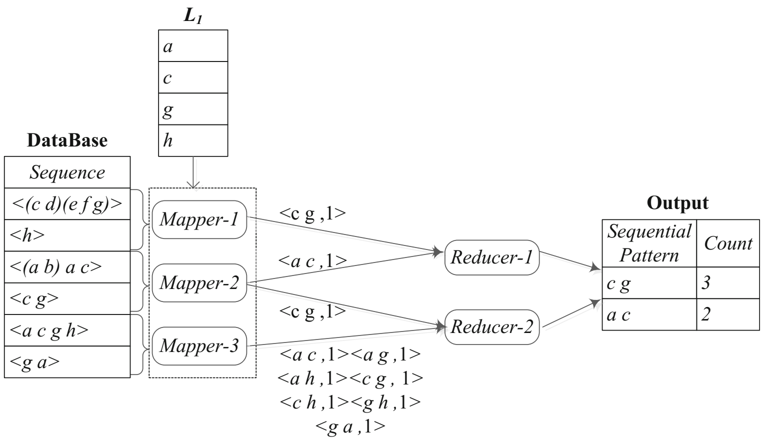


Figure 4 An execution example of the mining phase

In the database partition phase, GSP-S splits a sequence database into many database partitions. To balance load, one strategy is to split the sequence database into *n* equal-sized database partitions so that the total length of sequences in each database partition is almost the same. It is worth noting that the number of database partitions is preferably equal to the number of the computing nodes in a cluster in order to make the best of the cluster resources. If the database partitions are large and not fitted into the memory of a computing node, we split the sequence database so that the number of database partitions is several times than the number of the computing nodes until the database partitions can be fitted into the memory of a computing node.

The details of database partition are as follows. Firstly, all sequences in a sequence database are sorted by length in descending order using the quick sorting algorithm. Then, the front most *n* sequences form the initial *n* database partitions, one sequence per database partition. The total sequence length of each database partition is initialized with the length of the sequence it contains. Next, we build a min heap using the initial database partitions based on their respective lengths of the assigned sequences. Let $\Psi = \{D_1, D_2, D_3, \ldots, D_n\}$ be the list of database partitions after building the min heap. So, D_1 is assigned by the shortest sequence. We then add that non-assigned sequence into D_1, which has the maximum length among all non-assigned sequences. Next, we rearrange the min heap. Therefore, the database partition with the next minimum total length of sequences will be the root of the min heap. We pick up the database partition and assign a non-assigned sequence which has the maximum length among all the remaining sequences. The same procedure is continued until all the sequences in *D* are assigned to database partitions. Algorithm 4 presents the pseudo code of the database partition phase.

Algorithm 4 Database partition phase of GSP-S algorithm

```
Input: sequence database ($D$), the number of database partitions ($n$)
Output: database partitions ($D_i$)
1: DescendSort($D$);
2: Assign the front most $n$ sequences to form initial $n$ database partitions;
3: Build a min heap using the initial database partitions depending on their
   respective length of the assigned sequence;
4: for all non-assigned sequence $s$ in $D$ do
5:     Pick up the root node of the min heap say $D_i$;
6:     Assign $s$ to $D_i$ such that $s$ has the maximum length;
7:     Rearrange the min heap;
8: end for
```

Example Suppose GSP-S makes three partitions for the database in Table 1, such that the first database partition contains S_2 and S_3, the second database partition contains S_4 and S_6, and the third database partition contains S_1 and S_5 through our proposed database partition strategy, the input sequence database partitions are shown in Figure 5.

The time complexity of the database partition phase is $O(|D| \log |D| + |D| \log n)$, where *n* is the number of database partitions.

Proof Line 1 of Algorithm 4 requires $O(|D| \log |D|)$ time for sorting the sequences in *D* by length in descending order using the quick sorting algorithm. Lines 2–3 require $O(n \log n)$ time to build a min heap using *n* sequences. Lines 4–8 iterate $|D|$-n times

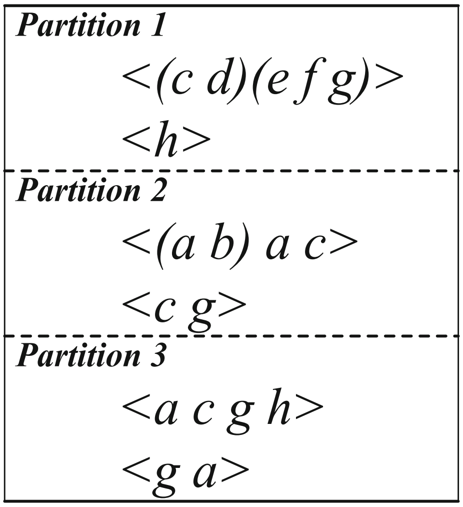


Figure 5 Input sequence database partitions after database partition phase

to assign the non-assigned sequences to database partitions. The time complexity of each iteration is $O(\log n)$. Therefore, the total time complexity of the database partition phase is $O(|D| \log |D| + |D| \log n)$.

Note that in the database partition phase, each sequence in D can be represented by its sequence-id and length, instead of copying the whole sequence when using the quick sorting algorithm and rearranging the min heap. Therefore, the time cost of the database partition phase is small. But the cost of the other two phases in GSP-S tends to occupy a larger proportion of the total algorithm execution time. Therefore, we ignore the time cost of the database partition phase when calculating the overall time complexity of GSP-S.

4.4 Analysis

The time complexity of GSP-S is $O(|D_i| \times l + |I| + \sum_{k \geq 2} (|L_k| \times |L_k| + |C_k| + |D_i| \times |C_k| + |C_k|))$, where $|D_i|$ is the number of sequences in the database partition D_i, and $|I|$ is the total number of items in the original database D.

Proof GSP-S adopts iterative MapReduce jobs. The time complexity of the first MapReduce job to find L_1 is $O(|D_i| \times l + |I|)$. In subsequent MapReduce jobs, the time complexity of generating the candidate sequence is $O(|L_k| \times |L_k| + |C_k|)$, the time complexity of scanning the database partition to compute the support count of the candidate sequences is $O(|D_i| \times |C_k|)$, and the time complexity of aggregating the support count of C_k is $O(|C_k|)$. Thus, the time complexity of GSP-S is $O(\text{GSP}-\text{S}) = O(|D_i| \times l + |I| + \sum_{k \geq 2} (|L_k| \times |L_k| + |C_k| + |D_i| \times |C_k| + |C_k|))$. The time complexity analysis shows that the time complexity of GSP-S is reduced by n time approximately compared with that of GSP, where n is the number of database partitions.

The IO overhead of GSP-S is $O(|D| \times l + \sum_{k \geq 1} (|L_k| \times k))$.

Proof In GSP-S, the first MapReduce job loads the database partitions generated in the database partition phase into the Spark RDDs from the HDFS, the IO overhead is $O(|D|)$. Subsequent MapReduce jobs read the database from the RDDs and store the obtained sequential patterns into the RDDs, and we persist the RDDs in memory, so it incurs almost no IO overhead (https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html). Finally, GSP-S writes all sequential patterns into the disk, and the IO overhead is $O(\sum_{k\geq 1}(|L_k| \times k))$. As a result, the total IO overhead of GSP-S is $O(|D| \times l + \sum_{k\geq 1}(|L_k| \times k))$.

The IO overhead is much lower than the iterative MapReduce jobs-based algorithms (i.e., DPSP [15], BIDE-MR [40], and MR-PrefixSpan [26]), since these iterative algorithms adopt multiple MapReduce jobs to implement parallel sequential pattern mining on Hadoop and each job needs to perform a read-write operation to the HDFS. However, reading the database from the RDDs and storing the obtained sequential patterns into the RDDs in GSP-S also lead to time and resource consumptions.

The network overhead of GSP-S is $O(|D| + n \times |I| + n \times \sum_{k\geq 1}(|L_k| + |C_k|))$.

Proof In the first MapReduce job, Master assigns n database partitions to n mappers, the network overhead is $O(|D|)$. Then, each mapper scans the database partition stored in this mapper to find L_1, it occurs no network overhead. Finally, mappers yield the $<item,1>$ key/value pairs, and the pairs with the same key are merged in a specific reducer, the network overhead is $O(n \times |I|)$. In the k-th MapReduce job ($k > 1$), Master first distributes $L_{k\text{-}1}$ to n mappers, the network overhead is $O(n \times |L_{k\text{-}1}|)$. Then, each mapper scans the database partition stored in this mapper to calculate the support count of C_k, it occurs no network overhead. Finally, mappers yield the $<c,1>$ key/value pairs for the candidate c in C_k, and the pairs with the same key are merged in a specific reducer, the network overhead is $O(n \times |C_k|)$. As a result, the total network overhead of GSP-S is $O(|D| + n \times |I| + n \times \sum_{k\geq 1}(|L_k| + |C_k|))$.

It is worth noting that we use the default Spark configurations and do not pay much attention to the network overhead problem, since Spark has a high-performance networking framework [3].

5 Parallelization of prefixspan

In this section, we first review the standard PrefixSpan algorithm [23, 24] and then propose our PrefixSpan-S algorithm. We also propose a projected database partition strategy to balance load among the computing nodes in a Spark cluster. Finally, the time complexity analysis, IO overhead and network overhead analysis of PrefixSpan-S are given.

5.1 Prefixspan algorithm

PrefixSpan is a pattern growth-based algorithm for mining sequential pattern. This algorithm constructs the projected database and then recursively mines them to discover all sequential patterns. The pseudocode for PrefixSpan is given as Algorithm 5.

Algorithm 5 PrefixSpan algorithm

```
Input: sequence database ($D$), $minsup$
Output: all sequential pattern( $L_k$)
1: Call PrefixSpan(<>,0,$S$)
2: procedure PrefixSpan($\alpha$,$l$,$S|_\alpha$)
3: The parameters are (1) $\alpha$ is a sequential pattern (2) $l$ is the length of $\alpha$ (3)
$S|_\alpha$ is $\alpha$-projected database if $\alpha$≠<>, otherwise it is the sequence database $D$
4: Scan $S|_\alpha$ once, find each frequent item $b$, such that:
5:     (a) $b$ can be assembled to the last element of $\alpha$ to form a sequential
pattern
6:     (b) <$b$> can be appended to $\alpha$ to form a sequential pattern
7:   for each frequent item $b$, append it to $\alpha$ to form a sequential pattern $\alpha'$
8:   for each $\alpha'$, construct $\alpha'$-projected database $S|_{\alpha'}$ and call
PrefixSpan($\alpha'$,$l$+1, $S|_{\alpha'}$)
```

The time complexity of PrefixSpan is $O(|D| \times l + m \times |S|_\alpha| \times l + m \times |S|_\alpha| \times l \times P)$, where $|D|$ is the number of sequences in D, l is the average length of the sequences, m is the number of building projected databases, $|S|_\alpha|$ is the number of sequences in $S|_\alpha$, and P is the average projecting time of an item.

Proof The time complexity of scanning the original sequence database to find L_1 is $O(|D| \times l)$. The time complexity of scanning the projected database to find length-1 sequential pattern is $O(|S|_\alpha| \times l)$. The time complexity of constructing the corresponding projected database $S|_\alpha$ for each α *is* $O(|S|_\alpha| \times l \times P)$. As a result, the total time complexity of PrefixSpan is $O(|D| \times l + m \times |S|_\alpha| \times l + m \times |S|_\alpha| \times l \times P)$.

The above analysis shows that the major time cost of PrefixSpan is the construction of projected databases. If the projected databases cannot entirely fit into the memory at the same time, PrefixSpan needs to write the projected databases into disk and read them from disk to perform physical projection. Since the projected databases are independent, we can assign them to different computing nodes for parallel mining of sequential patterns, which is the basis of our proposed PrefixSpan-S algorithm to be described next.

5.2 Prefixspan-S

Prefixspan-S has three important phases: database partition phase, support counting phase and projected database generation phase. The three phases are executed iteratively until no new sequential pattern is found. Figure 6 shows the framework of PrefixSpan-S. In the database partition phase, PrefixSpan-S splits the projected database into n database partitions based on our proposed database partition strategy to balance load. The details of this phase will be discussed in Section 5.3. The support counting phase generates the final sequential patterns through a MapReduce job. The projected database generation phase uses a MapReduce job to generate the projected database for each sequential pattern obtained in the support counting phase.

To reduce IO overhead and take advantage of cluster memory, the first MapReduce job loads the sequence database from the HDFS into the Spark RDDs, and subsequent MapReduce jobs read the projected databases from the RDDs and store the projected databases and sequential patterns generated in the support counting phase into the RDDs. Figure 7 shows the lineage graph for the RDDs in PrefixSpan-S. We present the details of the support counting phase and projected database generation phase below.

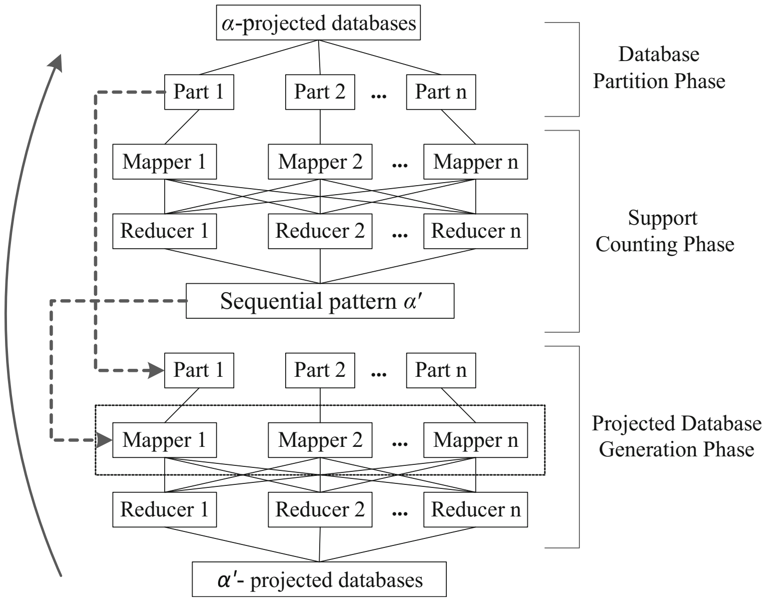


Figure 6 Framework of PrefixSpan-S algorithm

Support Counting Phase In this phase, we use a MapReduce job to find length-1 sequential patterns in $S|_\alpha$ and append them to α to form new sequential patterns α'. At the first execution of the support counting phase, $S|_\alpha$ is the original sequence database *D*. The MapReduce job is the same as the MapReduce job of the preparation phase in GSP-S. In further execution of the phase, each mapper invokes the first *flatMap*() function to read each sequence in the projected database partition $S_i|_\alpha$ generated in the database partition phase, and then invokes the other *flatMap*() function to split the suffix into items. Next, each mapper appends these items to α *and* yields the <α + item, 1 > key/value pairs. These key/value pairs with the same key are merged in a specific reducer. Finally, each reducer invokes the *reducebyKey*() function to

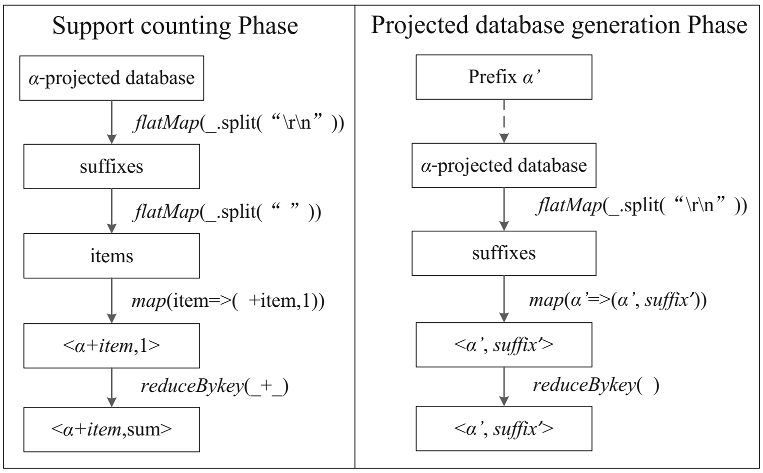


Figure 7 Lineage graph for the RDDs in PrefixSpan-S

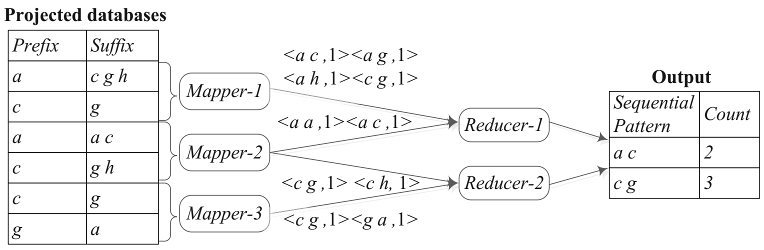


Figure 8 An execution example of the support counting phase

aggregate the support count of the items with prefix α and outputs $<\alpha +$ item, *sum* $>$ when the *sum* is not less than *minsup*. The pseudocode of the support counting phase is detailed in Algorithm 6.

Example Figure 8 illustrates an execution example of the support counting phase. Six suffixes generated in the above projected database generation phase are assigned to three mappers. The first mapper handles suffix $<c\ g\ h>$ prefixed with $<a>$, appends the items in the suffix to $<a>$, and outputs $<a\ c,\ 1>$, $<a\ g,\ 1>$ and $<a\ h,\ 1>$. Then the mapper handles suffix $<g>$ prefixed with $<c>$, appends $<g>$ to $<c>$, and outputs $<c\ g,\ 1>$. The other two mappers handle assigned suffixes and output key/value pairs in the same manner. The reducer aggregates the support count and outputs 2-sequential patterns $<a\ c,\ 2>$ and $<c\ g,\ 3>$.

Algorithm 6 Support counting phase of Prefixspan-S algorithm

```
Input: database partitions ( $S_i|_\alpha$), minsup,
Output: sequential patterns ($\alpha'$)
1: for all suffix in $S_i|_\alpha$ do
2:     flatMap(offset, suffix );
3:     for all item $b$ in suffix do
4:         output <$\alpha$+$b$,1>;
5:     end for
6: end for
7: reduceBykey(key=$\alpha$+ $b$, value=1);
8: sum=0;
9: for all key $\alpha$+ $b$ do
10:    for all value $v$ in $\alpha$+$b$'s value list do
11:        sum+=$v$;
12:    end for
13: end for
14: if sum $\geq$ minsup then
15:    output<$\alpha$+ $b$,sum>;
16: end if
```

Projected Database Generation Phase This phase constructs the corresponding projected database for the sequential patterns generated in the above MapReduce job. Note that sequences in the projected database of a prefix (e.g., prefix $<a\ c>$) is only the member of sequences in the projected database of the earlier prefix (i.e., prefix <*a*>). Therefore, the construction of α' -projected database does not refer to the original sequence database but is based on α-projected database (α' is a sequential pattern with prefix α). In this way, the size of the database sequences that must be parsed can be decreased drastically, because it refers only to the projected database of the earlier prefix [26]. As illustrated in the right-hand side of Figure 7, each mapper first invokes the *flatMap*() function to read suffix sequences in previous projected database partitions $S_i|_\alpha$ from the RDDs. Then a *map*() function is applied to compute the suffix for the prefix α' generated by the above MapReduce job, and yield the <α',*suffix*'> key/value pair. The process of generating suffix for prefix α' just reads suffix sequences in $S_i|_\alpha$ from the RDDs, *and* computes the suffix only for the sequential pattern α' with prefix α. The subsequence prefixed with the last item of sequential pattern α' *is* the suffix of α'. These key/value pairs with the same key are merged in a specific reducer. Finally, each reducer invokes the *reduceBykey*() function only to collect all suffix sequences and assemble them as the projected database. The pseudocode of the first MapReduce job is detailed in Algorithm 7.

Algorithm 7 Projected database generation phase of Prefixspan-S algorithm

```
Input: database partitions ( $S_i|_\alpha$), sequential patterns ($\alpha'$)
Output: projected database( $S|_{\alpha'}$)
1:  for all suffix s in $S_i|_\alpha$ do
2:     flatMap(offset,s);
3:     for all sequential pattern $\alpha'$
4:        if the last item of $\alpha'$ in s then
5:           String suffix=getsuffix($\alpha'$, s);
6:           output<$\alpha'$,suffix'>;
7:        end if
8:     end for
9:  end for
10: reduceBykey(key= $\alpha'$, value= suffix' );
11: for all value suffix' in value list do
12:    output<$\alpha'$, suffix' >;
13: end for
```

Example Figure 9 illustrates an execution example of the projected database generation phase. Six suffixes generated in the above projected database generation phase are assigned

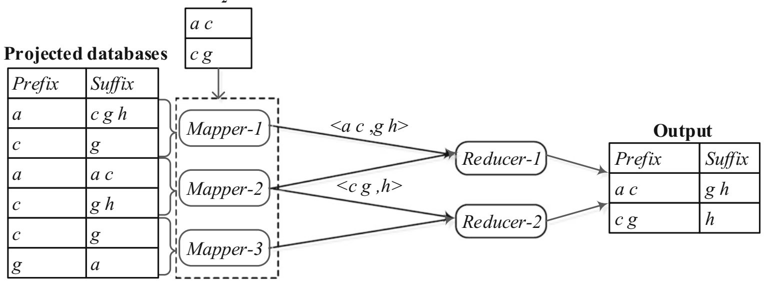


Figure 9 An execution example of projected database generation phase

to three mappers. The first mapper reads suffix <*c g h*> with prefix <*a*> from the RDDs, and computes the suffix only for those sequential patterns with prefix <*a*>. So the mapper only needs to compute the suffix for <*a c*>. The subsequence <*g h*> prefixed with the last item *c* of sequential pattern <*a c*> is the suffix of sequential pattern <*a c*>. Thus, the first mapper outputs <*a c*, *g h*> key/value pairs when reading the suffix sequence<*c g h*> with prefix <*a*> from the RDDs. Then the first mapper reads suffix <*g*> prefixed with <*c*> from the RDDs and computes the suffix only for the sequential pattern with prefix <*c*>. So, the mapper only needs to compute the suffix for <*c g*> and find that the subsequence prefixed with the first occurrence of the last item *g* of prefix <*c g*> is null, and output nothing. The other two mappers handle assigned suffixes and output key/value pairs in the same manner. The reducers only collect all key/value pairs and output <*a c*, *g h*> and < *c g*, *h*> key/value pairs.

5.3 Load balance

In order to balance load among the computing nodes of a Spark cluster, we first need to quantitatively measure the total computing load of each computing node. The computing cost of PrefixSpan-S mostly occurs in the projected database generation phase. So we pay particular attention to the *getsuffix()* function in the projected database generation phase. The function reads suffix sequences in the assigned projected database partition of prefix α, and computes the suffix only for the sequential pattern α' prefixed with α. That is to say, the computing load of *getsuffix()* function is proportional to the number of sequential patterns prefixed with α when reading the suffix sequence in α-projected database to compute the suffix for α'.

To balance load, for each sequential pattern α *in* L_k, we split the α-projected database into *n* partitions so that each database partition contains almost the same number of suffix sequences prefixed with α. That is, we want to disperse the suffix sequences prefixed with α to different mappers. While making these partitions, PrefixSpan-S filters the infrequent items. Note that if it is the first time to split the database, the database is the original sequence database; otherwise, it is the projected database. When it is the first time to split the database, the partition strategy is the same as the strategy described in the Section 4.3.

The time complexity of the database partition phase is $O(|D| \log |D| + |D| \log n + k \times |S|_\alpha|)$, where *n* is the number of database partitions, and *k* is the largest length of the sequential pattern.

Proof When it is the first time to split database, the partition strategy is the same as the strategy in the section 4.3, and the time complexity is $O(|D| \log |D|+| D| \log n)$. In the further partition phase, the time complexity is $O(|S|_\alpha|)$. Thus, the total time complexity of database partition phase is $O(|D| \log |D|+| D| \log n + k \times |S|_\alpha|)$.

The cost of the database partition phase is small and the cost of the other two phases in Prefixspan-S tends to occupy a larger proportion of the total algorithm execution time. Therefore, we ignore the time cost of the database partition phase when calculating the overall time complexity of PrefixSpan-S.

It is worth noting that Kessl [16] proposed a static load-balancing strategy for the parallel PrefixSpan algorithm on the distributed memory system. The differences between the strategy proposed by Kessl and ours are as follows. Our strategy is a dynamic load-balancing method, which splits the α-projected database into *n* partitions in each MapReduce job. The strategy first creates a sample of frequent sequences, and then use this sample for estimating the relative processing time of the algorithm, finally partitioned the sequence database. Compared with our strategy, the static load-balancing strategy cannot modify the load of each mappers at run time.

Example As shown in Figure 10, suppose Prefixspan-S makes three partitions for the projected databases of 1-sequential patterns <*a*>, <*c* > and < *g* > in Table 1, such that the first partition contains suffix <*c g h* > prefixed with <*a* > and suffix <*g*> prefixed with <*c*>, the second partition contains suffix <*a c* > prefixed with <*a* > and < *g h* > prefixed with <*c*>, and the third partition contains suffix sequence <*g* > prefixed with <*c* > and suffix <*a* > prefixed with <*g* > through our proposed database partition strategy.

5.4 Analysis

The time complexity of PrefixSpan-S is $O(|D_i| \times l + |I| + m \times |S_i|_\alpha| \times l + m \times |I| + m \times |S_i|_\alpha| \times l \times P)$, where $|S_i|_\alpha|$ is the number of sequences in $S_i|_\alpha$.

Proof The time complexity of the first MapReduce job to find L_1 is $O(|D_i| \times l + |I|)$. The time complexity of finding length-1 sequential pattern is $O(|S_i|_\alpha| \times l + |I|)$. Since the reducers of projected database generation phase only collect all the sequences and assemble them as the projected database, the time complexity can be ignored. The time complexity of constructing the corresponding project database $S|_\alpha$ for each α *is* $O(|S_i|_\alpha| \times l \times P)$. Thus, the total time complexity of the PrefixSpan-S algorithm is $O(|D_i| \times l + |I| + m \times |S_i|_\alpha| \times l + m \times |I| + m \times |S_i|_\alpha| \times l \times P)$. The time complexity analysis shows that the time complexity of PrefixSpan-S is reduced by *n* time approximately compared with that of PrefixSpan, where *n* is the number of projected database partitions.

The IO overhead of PrefixSpan-S is $O(|D| \times l + \sum_{k \geq 1} (|L_k| \times k))$.

Proof In PrefixSpan-S, the first MapReduce job loads the database partitions generated in the database partition phase into the Spark RDDs from the HDFS, and the IO overhead is $O(|D|)$.

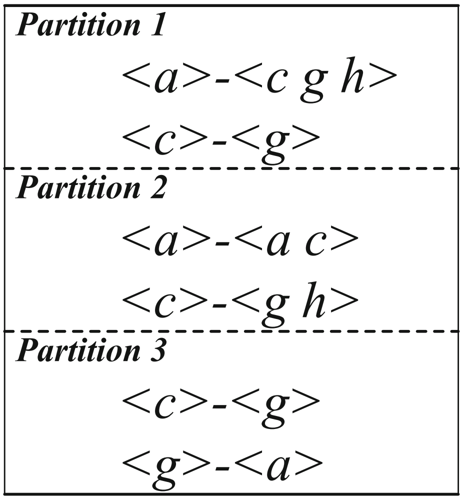


Figure 10 Projected database after partition phase

Since subsequent MapReduce jobs read the projected database from the RDDs and store the obtained sequential patterns and projected databases into the RDDs, it incurs almost no IO overhead. Finally, PrefixSpan-S writes all sequential patterns into disk, and the IO overhead is $O(\sum_{k\geq 1} (|L_k| \times k)$. As a result, the total IO overhead of PrefixSpan-S is $O(|D| \times l + \sum_{k\geq 1} (|L_k| \times k))$.

The network overhead of PrefixSpan-S is $O(|D| + n \times |I| + 2 \times n \times m \times |S_{\mathrm{i}}|_{\alpha}| + 2 \times n \times \sum_{k\geq 1} |L_k|)$.

Proof In the first MapReduce job, Master assigns n database partitions to n mappers, the network overhead is $O(|D|)$. Then, each mapper scans the database partition stored in this mapper to find L_1, it occurs no network overhead. Finally, mappers yield the $<item,1>$ key/value pair, and the pairs with the same key are merged in a specific reducer, the network overhead is $O(n \times |I|)$. In the second MapReduce job, Master first distributes L_1 to n mappers, the network overhead is $O(n \times |L_1|)$. Then, each mapper scans the database partition stored in this mapper to compute the suffix for the prefix sequence α in L_1, it occurs no network overhead. Finally, mappers yield the $<\alpha,suffix>$ key/value pair, and the pairs with the same key are merged in a specific reducer, the network overhead is $O(|S_{\mathrm{i}}|_{\alpha}|)$. In the $(2\ k$-1)-th MapReduce job $(k > 1)$, Master assigns n projected database partitions to n mappers, the network overhead is $O(|S_{\mathrm{i}}|_{\alpha}|)$. Then, each mapper scans the projected database partition stored in this mapper to find length-1 sequential pattern, it occurs no network overhead. Finally, mappers yield the $<\alpha +$ item, $1>$ key/value pairs, and the pairs with the same key are merged in a specific reducer, the network overhead is $O(n \times | L_{\mathrm{k}}|)$. In the $2\ k$-th MapReduce job $(k > 1)$, Master first distributes L_{k} to n mappers, the network overhead is $O(n \times |L_{\mathrm{k}}|)$. Then, each mapper scans the projected database partition stored in this mapper to compute the suffix for the prefix α' *in* L_{k}, it occurs no network overhead. Finally, mappers yield the $<\alpha',suffix'>$ key/value pair, and the pairs with the same key are merged in a specific reducer, the network overhead is $O(|S_{\mathrm{i}}|_{\alpha'}|)$. Therefore, the network overhead of PrefixSpan-S is $O(|D| + n \times |I| + 2 \times n \times m \times |S_{\mathrm{i}}|_{\alpha}| + 2 \times n \times \sum_{k\geq 1} |L_k|)$.

6 Expriment analysis

We evaluate the performance of GSP-S and PrefixSpan-S on a 10-node cluster. Each node possesses a single-core processor with 4 GB memory, and runs on the Ubuntu 14.04, on which Hadoop 2.7 and Spark 2.2 are installed. We set the number of Map and Reduce tasks by the default Hadoop parameter configurations and default Spark parameter configurations. Note that, if not specified, the number of data nodes is set as 4. This setting is followed by the similar work [37], which proposed a parallel frequent itemsets mining algorithm using MapReduce. The synthetic datasets as well as real-life datasets are used in our experiments.

(1) *Synthetic Datasets*: We employed two synthetic datasets (i.e., C20D10k and C20D150k) generated by the IBM Generator, which can be obtained from the SPMF repository [28].

(2) *Real life Datasets*: Since the real-life datasets from the SPMF repository are not very large, we apply GSP-S and Prefixspan-S to implement a data mining application for two very large taxi trajectory datasets. Since the trajectory data is the raw GPS data, we transform GPS position coordinates into trajectories expressed on a road segment level by calling the Baidu API Geocoding [5]. Two taxi trajectory data sets used in our experiments were generated by over 33,000 taxis in Beijing on November 1, 2012 and November 2, 2012. The datasets in these 2 days are hereinafter referred to as db1 and db2, respectively.

The characteristics of the four datasets are shown in Table 3.

6.1 Minimum support

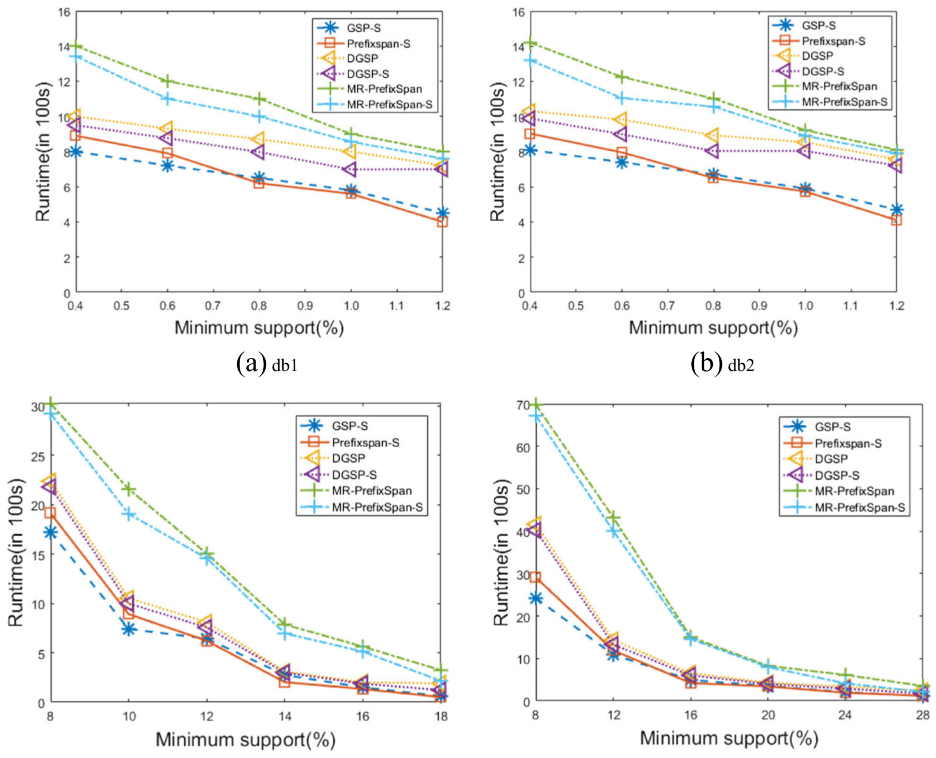
We analyze the runtime of the proposed GSP-S and PrefixSpan-S for different minimum support thresholds. We compare their runtime to DGSP [40] and MR-PrefixSpan [26] in Hadoop using the four datasets mentioned above. In addition, we adapt DGSP and MR-PrefixSpan in Spark and call them DGSP-S and MR-PrefixSpan-S, respectively. Figure 11a–d shows the results. As shown in Figure 11, a low minimum support degrades the mining performance of all algorithms. This is because the lower the minimum support is, the more sequential patterns the algorithms generate. It takes an increased amount of time to generate more sequential patterns. We observe that the proposed GSP-S and PrefixSpan-S perform significantly better than the compared algorithms. The reason for this is twofold. First, MR-PrefixSpan needs to read data from the HDFS in every MapReduce job, thus causing huge IO overhead. In contrast, GSP-S and PrefixSpan-S load the input dataset from the HDFS into the RDDs and then just read the data and intermediate results from the RDDs later, thus reducing IO overhead. Second, the efficient database partition strategies balance load well among computing nodes in the cluster, which also enables GSP-S and PrefixSpan-S to outperform other algorithms.

We also observe that PrefixSpan-S outperforms GSP-S when the minimum support is high. But when it comes to the low minimum support, GSP-S is superior to PrefixSpan-S. The performance trends are reasonable. The following reasons might cause the trends.

As stated above, the time complexity of PrefixSpan-S is $O(|D_i| \times l + |I| + m \times |S_\mathrm{i}|_\alpha| \times l + m \times |I| + m \times |S_\mathrm{i}|_\alpha| \times l \times P)$, *and the* network overhead of PrefixSpan-S is $O(|D| + n \times |I| + 2 \times n \times m \times |S_\mathrm{i}|_\alpha| + 2 \times n \times \sum_{k \geq 1} |L_k|)$. A great number of sequential patterns are generated, when the minimum support is low. PrefixSpan-S needs to construct the projected database for every sequential pattern, the cost is non-trivial [24]. The number of sequences in α-projected database partition is also large, the cost to find length-1 sequential patterns is also high. In addition, the network overhead $(O(|D| + n \times |I| + 2 \times n \times m \times |S_\mathrm{i}|_\alpha| + 2 \times n \times \sum_{k \geq 1} |L_k|))$

Table 3 Data parameters

Name	Number of sequences	Average length of sequences	Number of distinct items	Size of the dataset
C20D10K	10,000	20.0	192	0.81 MB
C20D150K	150,000	20.0	192	12.2 MB
db1	139,984	189.35	8799	1026 MB
db2	176,129	189.44	9277	1310 MB



(a) db1 (b) db2

(c) C20D10k (d) C20D150k

Figure 11 Runtime of algorithms with different minimum support on (**a**) db1 (**b**) db2 (**c**) C20D10k (**d**) C20D150k

increases dramatically and largely degrades the performance of PrefixSpan-S, when the value of minimum support decreases.

As stated above, the time complexity of GSP-S is $O(|D_i| \times l + |I| + \sum_{k\geq 2} (|L_k| \times |L_k| + |C_k| + |D_i| \times |C_k| + |C_k|))$, which shows that scanning the database partition D_i occupies a large proportion of the algorithm execution time. In addition, the low minimum support leads it to generate more candidate sequences. However, the network cost $(O(|D| + n \times |I| + n \times \sum_{k\geq 1} (|L_k| + |C_k|)))$ of GSP-S does not increase much dramatically than that of PrefixSpan-S. As a result, GSP-S outperforms PrefixSpan-S for low minimum support.

When the value of minimum support increases, for PrefixSpan-S, the number of sequential patterns decreases and so do the cost of constructing projected databases and network cost. But for GSP-S, the time spent in scanning the database is so dominant that the overall runtime of GSP-S does not decrease much despite the increase of minimum support.

6.2 Load balance

In this experiment, we test whether GSP-S and PrefixSpan-S can balance load to achieve optimal performance. We conduct this experiment using the taxi trajectory datasets mentioned above, because sequences in taxi trajectory datasets are of different length and highly skewed. Recall that our analysis shows the mining phase of GSP-S and the projected database generation phase of PrefixSpan-S are more sensitive to data distributions than the other phases of the algorithms. Therefore, we focus on the load-balancing performance of the mining phase of GSP-S and the projected database generation phase of PrefixSpan-S. We analyze the runtime of mappers in the second MapReduce job to test whether GSP-S and PrefixSpan-S are able to balance load. As discussed in Section 2, the existing parallel sequential pattern mining algorithms based on Hadoop are inefficient because they do not take load balance into consideration. We also compare the load-balancing performance of GSP-S and PrefixSpan-S with DGSP and MR-PrefixSpan. Since the time cost of the second MapReduce job in DGSP and MR-PrefixSpan tends to occupy a larger proportion of the total algorithm execution time, we analyze the runtime of mappers in the second MapReduce job of DGSP and MR-PrefixSpan.

We introduce the coefficient of variation (CV) to measure the load-balancing performance, because it eliminates the effect of measurement scale and measurement unit. CV = σ/u, where σ is the standard deviation of the runtimes of mappers in a MapReduce job, and u is the mean runtime of mappers in a MapReduce job. Figure 12 presents the CV value on db1 and db2.

As shown in Figure 12, GSP-S and PrefixSpan-S have lower CV than other algorithms. We also observe that load-balancing performance becomes more obvious when the value of minimum support decreases. In other words, GSP-S and PrefixSpan-S have a growing advantage in load-balancing performance when the load of each mapper increases.

We also conduct a more extensive experiment. We split the original sequence database using the Spark default settings and our database partition strategy, respectively. Figures 13 and 14 show the impact of our database partition strategy. The experiment results indicate that our database partition strategy brings significant improvements in performance, because our load-balancing strategies significantly shorten the execution time of GSP-S and PrefixSpan-S. To be specific, our database partition strategies reduce the runtime of GSP-S at least by 18.97% (=(6.9–5.8)/5.8), and reduce the runtime of PrefixSpan-S at least by 21.85% (=(7.92–6.5)/6.5).

6.3 Speedup

In this experiment, we study the speedup performance of GSP-S and PrefixSpan-S. Speedup means the ratio of the runtime on a single computing node to the runtime for an identical data

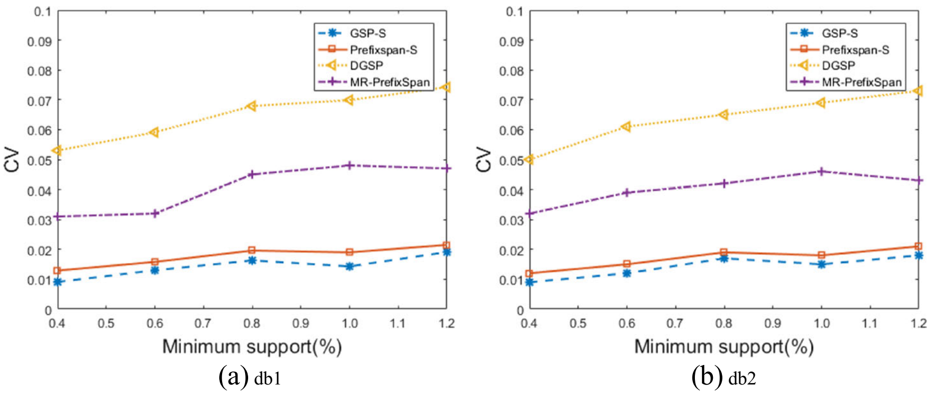
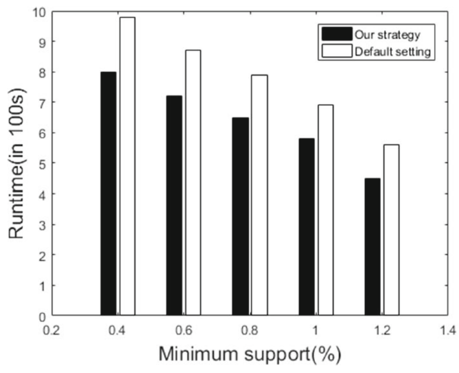
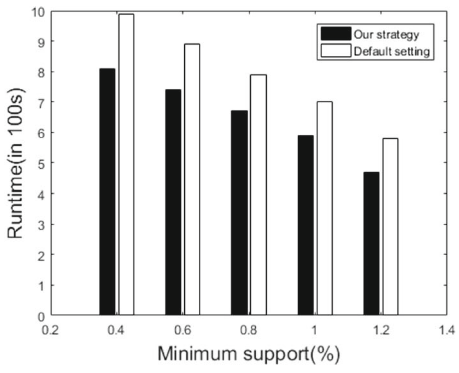


Figure 12 Coefficient of variation of runtimes with different minimum support on (**a**) db1 (**b**) db2



(a) db1

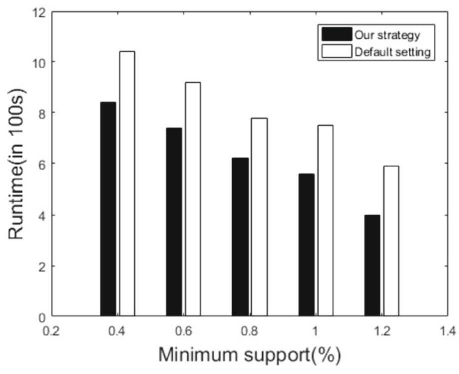


(b) db2

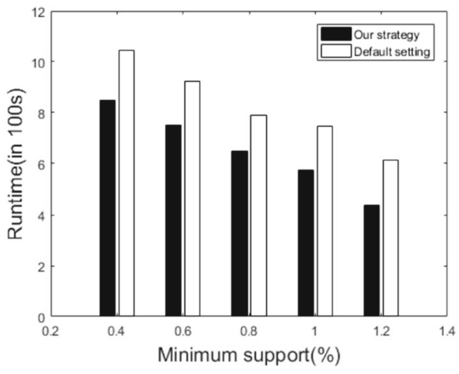
Figure 13 Runtime of GSP-S with the default settings and our database partition strategy on (**a**) db1 (**b**) db2

set on the cluster. The *speedup* = T_s/T_p, where T_s is the runtime of the serial algorithm on a single computing node and T_p is the runtime of the parallel algorithm on p data nodes for the same dataset. We study the speedup performance by taking the ratio of baseline (GSP or PrefixSpan) runtime on a single computing node to the runtime of GSP-S and PrefixSpan-S on a Spark cluster for the same dataset. We study the speedup performance of GSP-S and PrefixSpan-S by increasing the number of data nodes from 1 to 9 in the Spark cluster. We conduct the experiment on db1 and db2, because these two datasets contain more sequence than C20D10k and C20D150k. Here, we set the minimum support as 1% (little difference of the scalability performance was observed with other minimum supports).

In Figure 15, we plot the speedup gained by the algorithms on db1 and db2. Recall from Sections 4.4 and 4.5 the time complexity analysis shows that GSP-S and PrefixSpan-S can achieve linear speedup compared with GSP and PrefixSpan, respectively. From these plots, we observe that the speedup of GPS-S and PrefixSpan-S is close to the linear speedup. The reason they cannot be linear is that the communication cost between mappers and reducers goes up when more computing nodes are added. As the number of computing nodes increases, the size of the assigned database partition per computing node decreases and so does the computation cost for each computing node. But the communication cost among the computing nodes



(a) db1



(b) db2

Figure 14 Runtime of PrefixSpan-S with the default settings and our database partition strategy on (**a**) db1 (**b**) db2

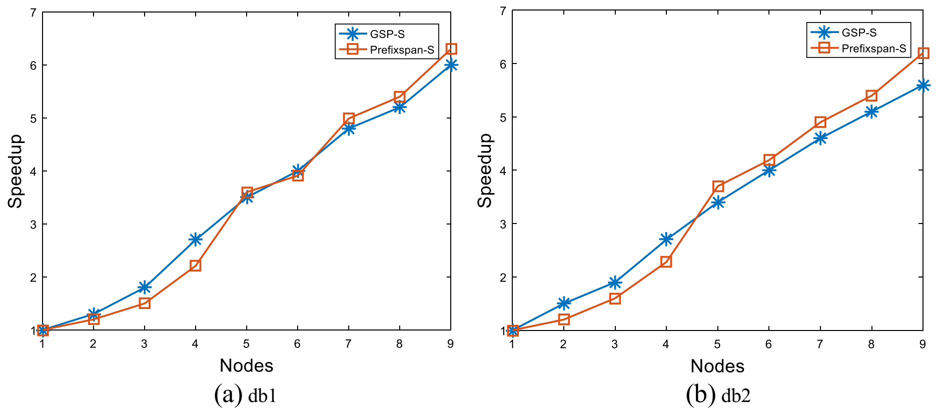


Figure 15 Speedup performance of GSP-S and PrefixSpan-S on (**a**) db1 and (**b**) db2

degrades the overall performance of algorithms. In general, GSP-S and PrefixSpan-S gain significant speedup on a Spark cluster.

6.4 Scalability

In this experiment, we study the scalability performance of GSP-S and PrefixSpan-S when the size of input sequence database grows. We generate five synthetic datasets each containing 50 k to 250 k input sequences with an increment of 50 k. Here, we set the minimum support as 10% (little difference of the scalability performance was observed with other minimum supports), and record the runtime of the evaluated algorithms for different sizes of input sequence database.

As shown in Figure 16, the runtime of GSP-S and PrefixSpan-S goes up when the size of input sequence database increases. The reason is twofold. First, the cost of loading the input dataset from the HDFS into the RDDs grows. Second, the increased input dataset leads to a

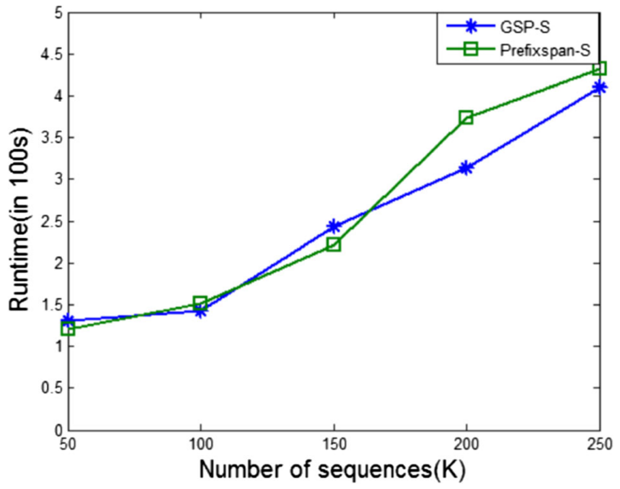


Figure 16 Scalability of GSP-S and PrefixSpan-S when the size of input sequence database increases

longer database scanning time of GSP-S. For PrefixSpan-S, the time cost of constructing projected databases grows along with the size of the input dataset. Nevertheless, the overall runtime increases at a sub-linear rate as the size of the input sequence database increases because of the efficiency of our in-memory computation strategy and load balance strategies.

6.5 Analysis

The aforementioned experiment results conclude that GSP-S and PrefixSpan-S outperform the existing approaches (see Figures 11 and 12). The reason for this is also twofold. First, GSP-S and PrefixSpan-S load the input dataset from the HDFS into the RDDs and then just read the data and intermediate results from the RDDs later, thus reducing IO overhead. Second, the efficient database partition strategies balance load well among computing nodes in the cluster, which also enables GSP-S and PrefixSpan-S to outperform other algorithms. More importantly, we also observe that GSP-S outperforms PrefixSpan-S for the low minimum support. But when it comes to the high minimum support, PrefixSpan-S is superior to GSP-S (see Figure 11). Our findings suggest that a wise choice can be made between GSP-S and PrefixSpan-S, depending on the user-specified minimum support threshold.

7 Conlusion and future work

In this paper, we propose two scalable and parallel sequential pattern mining algorithms based on Spark, called GSP-S and PrefixSpan-S. Compared with the existing parallel algorithms, GSP-S and PrefixSpan-S overcome the high IO overhead problem by adopting in-memory computation. In addition, we improve the performance of GSP-S and PrefixSpan-S by balancing workload among computing nodes in the cluster. Experimental results show the high performance of GSP-S and PrefixSpan-S in terms of load-balancing, speedup and scalability. Moreover, our new findings suggest that users can make a sensible choice between GSP-S and PrefixSpan-S depending on the user-specified minimum support threshold.

In the further, we plan to explore other load-balancing strategies to improve the performance even further. We also plan to apply the proposed algorithms suitable for mining uncertain data [17], and other related applications.

Acknowledgement The authors would like to acknowledge the support provided by the National Key Research and Development Program of China (No. 2017YFB1400602 and No.2016YFB0800401), the grands of the National Natural Science Foundation of China (61572374, U163620068, U1135005, 61572371) and Academic Team Building Plan for Young Scholars from Wuhan University (WHU2016012).

References

1. Aggarwal, C.-C., Han, J.: Frequent pattern mining. Springer.
2. Agrawal, R., Srikant, R.: Mining sequential pattern. In: 11th International Conference on Data Engineering, pp. 3–14. IEEE(1995)

3. Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Zaharia, M.: Scaling spark in the real world: performance and usability. Proceedings of the VLDB Endowment. **8**(12), 1840–1843 (2015)
4. Ayres, J., Gehrke, J., Yiu, T., et al: Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 429–435(2002)
5. Baidu Geocoding: (2016). Available from: http://lbsyun.baidu.com/
6. Chen, C.-C., Tseng, C.-Y., Chen, M.-S.: Highly scalable sequential pattern mining based on mapreduce model on the cloud. In: 2013 I.E. International Congress on Big Data, pp. 310–317. IEEE (2013)
7. Hu, Y., Cheng-Kui Huang, T.: Knowledge discovery of weighted RFM sequential patterns from customer sequence databases. J. Syst. Softw., vol. 86, no. 3, pp. 779–788(2013)
8. Cong, S., Han, J., Padua, D.: Parallel mining of closed sequential patterns. In: KDD '05 Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp. 562–567(2005)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM. **51**(1), 107–113 (2008)
10. Fournier-Viger, P., Wu, C.-W., Tseng, V.-S.: Mining maximal sequential patterns without candidate maintenance. In: International Conference on Advanced Data Mining and Applications, Springer, Berlin, Heidelberg, pp. 169–180(2013)
11. Guan, E.-Z., Chang, X.-Y., Wang, Z., Zhou, C.-G.: Mining maximal sequential patterns.In: Proc of the Second Int'l Conf. Neural Networks and Brain, pp. 525–528(2005)
12. Gurainik, V., Garg, N., Karypis, G.: Parallel tree projection algorithm for sequence mining. In: 7th International Euro-Par Conference on Parallel Processing, pp. 310–320(2001)
13. Hadoop Website, http://hadoop.apache.org/
14. Han, J., Pei, J., Mortazavi-Asl, B., et al.: FreeSpan: frequent pattern-projected sequential pattern mining. In: Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 355–359(2000)
15. Huang, J., Lin, S., Chen, M.: DPSP: distributed progressive sequential pattern mining on the cloud. Advances in Knowledge Discovery and Data Mining. 27–34 (2010)
16. Kessl, R.: Probabilistic static load-balancing of parallel mining of frequent sequences. IEEE Trans. Knowl. Data Eng. **28**(5), 1299–1311 (2016)
17. Leung, C.-K.-S., MacKinnon, R.-K., Jiang, F.: Finding efficiencies in frequent pattern mining from big uncertain data. World Wide Web. **20**(3), 571–594 (2017)
18. Li, C., Yang, Q., Wang, J., Li, M.: Efficient mining of gap-constrained subsequences and its various applications. ACM Trans. Knowl. Discov. Data. **6**(1), 2:1–2:39 (2012)
19. Liao, V.-C.-C., Chen, M.-S.: DFSP: a depth-first SPelling algorithm for sequential pattern mining of biological sequences. Knowl. Inf. Syst. **38**(3), 623–639 (2014)
20. Liu, C., Yao, L., Li, J., Zhou, R., He, Z.: Finding smallest k-compact tree set for keyword queries on graphs using mapreduce. World Wide Web. **19**(3), 499–518 (2016)
21. Lu, S., Li, C.: AprioriAdjust: an efficient algorithm for discovering the maximum sequential patterns. In: Proc. 2nd Int'l Workshop Knowl. Grid and Grid Intell(2004)
22. Luo, C., Chung, S. M.: Efficient mining of maximal sequential patterns using multiple samples. In: Proceedings of the 2005 SIAM International Conference on Data Mining, Society for Industrial and Applied Mathematics, pp. 415–426(2005)
23. Pei, J.: Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Computer Society. **16**(11), 1424–1440 (2004)
24. Pei, J., Han, J., Pinto, H.: PrefixSpan: mining sequential pattern efficiently by prefix-projected pattern growth. In: 17th international conference on data. Engineering. 215–224 (2001)
25. Pinto, H., Han, J., Pei, J., Wang, K., Chen, Q., Dayal, U.: Multi-dimensional sequential pattern mining. In CIKM Conference, pp. 81–88(2001)
26. Sabrina, P.-N.: Miltiple MapReduce and derivative projected database: new approach for supporting prefixspan scalability. In: 2015 I.E. International Conference on Data and Software Engineering, pp. 148–153. IEEE (2015)
27. Shintani, T., Kitsuregawa, M.: Mining algorithms for sequential patterns in parallel: hash based approach. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, Berlin, Heidelberg, pp. 283–294(1998)
28. SPMF: http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php
29. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. Advances in Database Technology — EDBT '96. **1057**, 1–17 (1996)
30. Wang, X.: Parallel sequential pattern mining by transcation decompostion. The International Conference on Fuzzy Systems and Knowledge Discovery. **4**, 1746–1750 (2010)

31. Wang, J., Han, J.: Bide:Efficientminingoffrequentclosedsequences. In: 20th International Conference on Data Engineering, pp. 79–90. IEEE (2004)
32. Wang, J., Han, J., Li, C.: Frequent closed sequence mining without candidate maintenance. TKDE. **19**(8), 1042–1056 (2007)
33. Wang, T., Zhang, D., Zhou, X., et al.: Mining personal frequent routes via road corner detection. IEEE Trans. Syst. **46**(4), 445–458 (2016)
34. Wei, Q.-Y., Liu, D., Duan, S.-L.: Distributed PrefixSpan algorithm based on MapReduce. In: 2012 International Symposium on Information Technology in Medicine and Education, pp. 901–904(2012)
35. Wu, C., Lai, C., Lo, Y.: An empirical study on mining sequential patterns in a grid computing environment. Expert Syst. Appl. **39**(5), 5748–5757 (2012)
36. Xin, J., Wang, Z., Chen, C., Ding, L., Wang, G., Zhao, Y.: ELM∗: distributed extreme learning machine with MapReduce. World Wide Web. **17**(5), 1189–1204 (2014)
37. Xun, Y., Zhang, J., Qin, X.: FiDoop: parallel Mining of Frequent Itemsets Using MapReduce. IEEE Transactions on Systems, Man, and Cybernetics: Systems. **46**(3), 313–325 (2016)
38. Yan, X., Han, J., Afshar, R.: Clospan:Mining closed sequential patterns in large datasets. In: SDM Conference, pp. 166–177(2003)
39. Yu, C.-C., Chen, Y.-L.: Mining sequential patterns from multidimensional sequence data. IEEE Trans. Knowl. Data Eng. **17**(1), 136–140 (2005)
40. Yu, D., Wu, W., Zheng, S., Zhu, Z.: BIDE-based ParallelMining of frequent closed sequences with MapReduce. In: Proceedings of the 12th International Conference on Algorithms and Architecturesfor Parallel Processing, pp.177–186(2012)
41. Yu, X., Liu, J., Ma, C., Li, B.: A MapReduc reinforeced distirbuted sequential pattern mining algorithm. Algorithms and Architectures for Parallel Processing. **9529**, 183–197 (2015)
42. Zaharia, M., et al.: Spark: cluster computing with working sets. HotCloud, pp. 10–10(2010)
43. Zaharia, M., et al: 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, USENIX Association(2012)
44. Zaki, M.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. **41**(2), 31–60 (2001)
45. Zaki, M.J.: Parallel sequence mining on shared-memory machines. J. Parallel Distrib. Comput. **61**(3), 401–426 (2001)
46. Zhang, C., Hu, K., Liu, H.: FMGSP: an efficient method of mining global sequential pattern. In: 4th International Conference on Fuzzy Systems and Knowledge Discovery, pp. 761–765(2007)
47. Zheng, Z., Wei, W., Liu, C., et al.: An effective contrast sequential pattern mining approach to taxpayer behavior analysis. World Wide Web-internet & Web Information Systems. **19**(4), 633–651 (2016)