DPSP: Distributed Progressive Sequential Pattern Mining on the Cloud

Jen-Wei Huang[1], Su-Chen Lin[2], and Ming-Syan Chen[2]

[1] Yuan Ze University, Taiwan
jwhuang@saturn.yzu.edu.tw
[2] National Taiwan University, Taiwan

Abstract. The progressive sequential pattern mining problem has been discussed in previous research works. With the increasing amount of data, single processors struggle to scale up. Traditional algorithms running on a single machine may have scalability troubles. Therefore, mining progressive sequential patterns intrinsically suffers from the scalability problem. In view of this, we design a distributed mining algorithm to address the scalability problem of mining progressive sequential patterns. The proposed algorithm DPSP, standing for Distributed Progressive Sequential Pattern mining algorithm, is implemented on top of Hadoop platform, which realizes the cloud computing environment. We propose Map/Reduce jobs in DPSP to delete obsolete itemsets, update current candidate sequential patterns and report up-to-date frequent sequential patterns within each POI. The experimental results show that DPSP possesses great scalability and consequently increases the performance and the practicability of mining algorithms.

1 Introduction

Based on the earlier work [7], the sequential pattern mining problem [1] can be categorized as three classes according to the management of corresponding databases. They are static sequential pattern mining, incremental sequential mining and progressive sequential pattern mining. It is noted that the progressive sequential pattern mining is known as a general model of the sequential pattern mining. The static and the incremental sequential pattern mining can be viewed as special cases of the progressive sequential pattern mining. The progressive sequential pattern mining problem can be described as "Given an interesting time period called period of interest (POI) and a minimum support threshold, find the complete set of frequent subsequences whose occurrence frequencies are greater than or equal to the minimum support times the number of sequences having elements in the current POI in a progressive sequence database." In fact, mining progressive sequential patterns intrinsically suffers from the scalability problem. In this work, we propose a distributed data mining algorithm to address the scalability problem of the progressive sequential pattern mining. The proposed algorithm DPSP, which stands for Distributed Progressive Sequential Pattern mining algorithm, is designed on top of Hadoop platform [6], which implements Google's Map/Reduce paradigm [5].

We design two Map/Reduce jobs in DPSP. At each timestamp, the candidate computing job computes candidate sequential patterns of all sequences and updates the summary of each sequence for the future computation. Then, using all candidate sequential patterns as the input data, the support assembling job accumulates the occurrence frequencies of candidate sequential patterns in the current POI and reports frequent sequential patterns to users. Finally, all up-to-date frequent sequential patterns in the current POI are reported. DPSP not only outputs frequent sequential patterns in the current POI but also stores summaries of candidate sequential patterns at the current timestamp. As time goes by, DPSP reads back summaries of all sequences and combine them with newly arriving itemsets to form new candidate sequential patterns at the new timestamp. Obsolete candidate sequential patterns are deleted at the same time. DPSP is thus able to delete obsolete itemsets, update summaries of all sequences and report up-to-date frequent sequential patterns. It is noted that DPSP does not need to scan the whole database many times to gather occurrence frequencies of candidate sequential patterns. DPSP, instead, reads newly arriving data and the summary of each sequence once. In addition, DPSP utilizes cloud computing techniques. It is easy to scale out using Hadoop platform to deal with huge amounts of data. The experimental results show that DPSP can find progressive sequential patterns efficiently and DPSP possesses great scalability. The distributed scheme not only improves the efficiency but also consequently increases the practicability.

The rest of this work is organized as follows. We will derive some preliminaries in Section 2. The proposed algorithm DPSP will be introduced in Section 3. Some experiments to evaluate the performance will be shown in Section 4. Finally, the conclusion is given in Section 5.

2 Related Works

After the first work addressing the sequential pattern mining problem in [1], many research works are proposed to solve the static sequential pattern mining problem [2], and the incremental sequential pattern mining problem [10]. As for the progressive sequential pattern mining problem, new data arrive at the database and obsolete data are deleted at the same time. In this model, users can focus on the up-to-date database and find frequent sequential patterns without being influenced by obsolete data. To deal with a progressive database efficiently, a progressive algorithm, Pisa, is proposed in [7]. However, traditional algorithms running on a single processor struggle to scale up with huge amount of data. In view of this, many researchers work on distributed and parallel data mining algorithms [4] [3] [12] [9] [11] [8]. In recent days, many researchers and corporations work on developing the cloud computing technology, which utilizes clusters of machines to cope with huge amount of data. The platform allows developers to focus on designing distributed algorithms whereas routine issues like data allocation, job scheduling, load balancing, and failure recovery can be inherently handled by the cloud computing framework. Hadoop [6] is an open

```
Algorithm DPSP
Input: itemsets of all sequences arriving at the current timestamp
1. while(there is new data at timestamp t){
2.   CandidateComputingJob;
3.   SupportAssemblingJob;
4.   t = t + 1;
5.   output frequent sequential patterns;
6. }end while
output: frequent sequential patterns at each POI with their supports
End
```

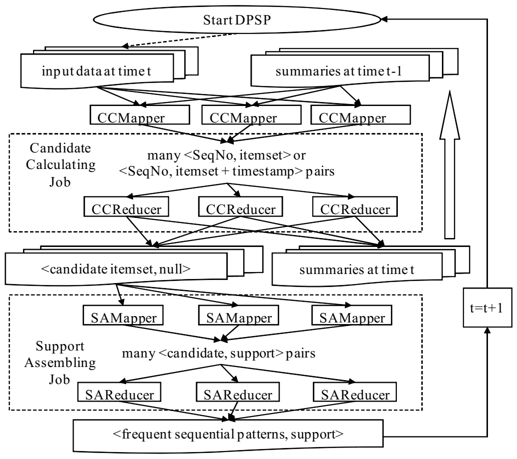

Fig. 1. Algorithm DPSP and system model

source project aiming at building a cloud infrastructure running on large clusters, which implements Google's Map/Reduce paradigm [5]. By means of the map function, the application can be divided into several fractions. Each fraction is assigned to a single node in large clusters and executed by the node. After the execution, the reduce function merges these partial results to form the final output. As such, developers need only to design a series of Map/Reduce jobs to split data and merge results.

3 Distributed Progressive Sequential Pattern Mining

We utilize Hadoop platform to design a distributed algorithm for the progressive sequential pattern mining. The proposed algorithm is named as Distributed Progressive Sequential Pattern mining algorithm, abbreviated as DPSP. In essence, DPSP consists of two Map/Reduce jobs, the candidate computing job and the support assembling job. As shown in the left of Figure 1, for each timestamp, the candidate computing job reads input data, which arrives at timestamp t, of all sequences. Itemsets from different sequences are distributed to different nodes in the cloud computing environment. Each node in the cloud computes candidate sequential patterns of each sequence within the current POI. Meanwhile, the candidate computing job also updates the summary for each sequence. Obsolete data are deleted in the candidate computing job and the up-to-date candidate sequential patterns are output. Then, support assembling job reads all candidate sequential patterns as input data. Different candidate sequential patterns are distributed to different nodes. Each node accumulates the occurrence frequencies of candidate sequential patterns and reports frequent sequential patterns whose supports are no less than the minimum support threshold in the current POI to users. When time goes to the next timestamp, DPSP keeps executing these Map/Reduce jobs. As such, DPSP is able to report the most up-to-date frequent sequential patterns in each POI.

The system model of DPSP is shown in right of Figure 1. The upper part is the candidate computing job while the support assembling job is at the lower part. In the candidate computing job, input data at timestamp t and the candidate set summaries at timestamp t-1 are split and transferred to several CCMappers. CCMapper generates many pairs of <sequence number, input itemset>. Then, pairs with the same sequence number are sent to the same CCReducer. CCReducer computes candidate sequential patterns of the given sequence and outputs pairs of <candidate sequential patterns, null>. In addition, CCReducer updates the summary of each sequence and deletes obsolete data at the same time. Candidate set summaries at the current timestamp are output for the computation at the next timestamp as well. Next, each SAMapper in the support assembling job reads input data and accumulates local occurrence frequencies for each candidate sequential patterns. SAMapper generates pairs of <candidate sequential pattern, local supports of the candidate> as outputs. Then, the pairs containing the same candidate sequential pattern are sent to the same SAReducer. SAReducer aggregates supports of the same candidate sequential pattern and outputs those frequent patterns in the current POI. After the computation at the timestamp, t, DPSP moves to the next timestamp, t+1.

3.1 Candidate Computing Job

The objective of the candidate computing job is to compute all candidate sequential patterns from all sequences within the current POI as shown in Figure 2. In CCMapper, itemsets of all sequences arriving at the current timestamp and the candidate set summaries at the previous timestamp are used as input data. As shown in lines 2 to 3 of CCMapper, if CCMapper reads the input from candidate set summaries, CCMapper generates <sequence number, candidate itemset with the corresponding timestamp> pairs. On the other hand, if CCMapper reads the input data from a sequence, CCMapper outputs <sequence number, arriving itemset> pairs as shown in lines 4 to 5. These output pairs are distributed to CCReducers as their inputs. Pairs with the same key are sent to the same CCReducer. By means of the summary at the previous timestamp and the arriving itemset at the current timestamp, each CCReducer is able to generate candidate sequential patterns of each sequence in the current POI. In line 2 of CCReducer, the multiple output variable is used to output candidate set summary at the current timestamp for the future computation. In lines 6 to 15, CCReducer enumerates each value in the receiving pairs. If the value is a candidate set summary at the previous timestamp, CCReducer puts the candidate into cand_set. In lines 9 to 10, if the timestamp is bigger than the start time of the current POI, which means this candidate will still be valid at the next timestamp, CCReducer outputs the candidate in the summary of the current timestamp for the computation at the next timestamp. In lines 11 to 12, if the candidate contains more than 1 item, the candidate is put in the result set as

```
CandidatesComputingJob:

CCMapper
input: itemsets of all sequences arriving at the current
timestamp OR candidate set summaries at the previous
timestamp
map:
1. var data = read input data;
2. if(data contains timestamp){ // it is a candidate
3.   output <data.sequenceNo, data.itemset+ data.time>;
4. else  // it is an arriving itemset
5.   output <data.sequenceNo, data.itemset>;

CCReducer
configure:
1. var start_time = current timestamp - POI;
2. var mo.output<sequenceNo, itemset + timestamp>;
   // multiple output to output summary
reduce(in_key, in_values):
3. var input; // used to store input data
4. var cand_set; // used to store candidate set summary at
the previous timestamp
5. var result<itemset>; // used to store distinct results
6. for(each value in in_values){
7.   if(value is a summary){
8.     cand_set.put(value.time, value.itemset);
9.     if(value.time > start_time)
10.      mo.output<in_key, candidate + candidate.time>;
11.    if(value.itemset.size > 1)
12.      result.put(value.itemset);
13.  }else // it is input data
14.    input = get the only one data at current timestamp;
15.}end for
16.for(each combination of items in input.itemset){
17.  for(each candidate in cand_set){
18.    var new_cand = append combination to candidate;
19.    result.put(new_cand);
20.    if(candidate.time > start_time)
21.      mo.output<in_key, new_cand + candidate.time>;
22.  }end for
23.  mo.output<in_key, combination + current time>;
24.}end for
25.for(each itemset in result)
26.  output<itemset, null>;
output: sequence number and candidate sequential itemsets
with timestamps

SupportsAssemblingJob:

SAMapper
input: candidate seuqential patterns of all sequences
configure:
1. var localMap<itemset, support>;
   // used to locally aggregate supports.
map:
2. var data = read input data;
3. if(data.itemset is in localMap)
4.   localMap(itemset).support++;
5. else
6.   insert <itemset, 1> into localMap;
close:
7. for(each <itemset, support> pair in localMap)
8.   output <itemset, support>;

SAReducer
reduce(in_key, in_values):
1. var count = 0;
2. for(each value in in_values)
3.   count += value;
4. if(count >= minimum support)
5.   output<in_key, count>;
output: frequent sequential patterns with their supports
```

Fig. 2. Candidates Computing Job and Supports Assembling Job

a candidate sequential pattern. In lines 13 to 14, if the value is the arriving itemset of a sequence, CCReducer stores the input itemset for the generation of new candidate itemsets in the following lines. It is noted that there is only one newly arriving itemset of a specific sequence number at a timestamp.

CCReducer has to compute all combinations of items in the arriving itemset in order to generate the complete set of different sequential patterns. For example, if the incoming itemset is (ABC), all combinations for generating candidate sequential patterns are A, B, C, (AB), (AC), (BC), and (ABC). In lines 17 to 22, CCReducer first appends each combination to each candidate itemset in the cand_set summary to form new candidate sequential patterns. Then, the newly generated candidate sequential pattern is put into the result set as an output in line 19. Meanwhile, if the timestamp of the candidate sequential pattern is bigger than the start time of the current POI, the newly generated candidate itemset is put in the summary of the current timestamp for further computation at the next timestamp in lines 20 to 21. Note that the candidate itemsets whose timestamps equal to the start time are not stored. In other words, the obsolete data at the next timestamp are pruned away. In addition to the newly generated candidates, CCReducer stores each combination with the current timestamp in the summary at the current timestamp in line 23. The summary of the current timestamp will be used to compute candidate sequential patterns at the next timestamp. Finally, all candidate itemsets in the result set are output as <candidate itemset, null> pairs in lines 25 to 26. After the collection of output pairs of each CCReducer, the candidate computing job has dealt with all incoming itemsets at the current timestamp, generated candidate sequential patterns of all sequences in the current POI, and updated candidate set summaries of all sequences for the computation at the next timestamp.

3.2 Support Assembling Job

As shown in Figure 2, the support assembling job calculate supports for each candidate sequential patterns. The support assembling job reads all candidate sequential patterns from the outputs of the candidate computing job. SAMapper utilizes a local map to aggregate occurrence frequencies of different candidate sequential patterns locally in lines 2 to 6 and outputs <candidate sequential pattern, its local supports> pairs in lines 7 to 8. Pairs with the same candidate sequential pattern are sent to the same SAReducer. In lines 2 to 3 of SAReducer, SAReducer accumulates supports of the same candidate sequential pattern again and gathers the final supports. For those candidate sequential patterns whose supports are no less than the minimum support threshold, SAReducer reports them as frequent sequential patterns in the current POI in lines 4 to 5. Then, DPSP algorithm moves to the next timestamp and repeats these Map/Reduce jobs.

4 Performance Evaluation

4.1 Experimental Designs

To assess the performance of DPSP, we conduct several experiments to evaluate the performance and the effects of input parameters. DPSP is implemented in Java and runs on top of Hadoop version 0.19.1. Hadoop cluster consists of 13 nodes and each node contains 2 intel Xeon(TM) CPU 3.20GHz, 2GB RAM and 32GB SCSI hard disk. The synthetic datasets are generated the same as [7]. In our experiments, every point in the figures is the total execution time of 40 timestamps and the POI is set as 10 timestamps unless specified otherwise. The minimum support threshold is set to 0.02 and there are 10000 different items in the synthetic datasets.

4.2 Experimental Results

First, we examine the performance of DPSP with large numbers of sequences as shown in Figure 3. Note that both X-axis and Y-axis are in log scale in (a). The total execution time does not vary a lot when the number of sequences is

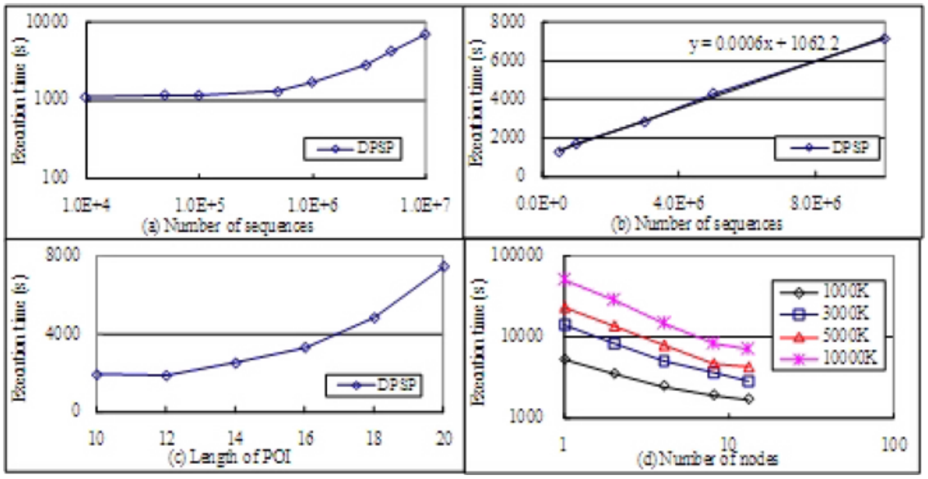

Fig. 3. Experiments

smaller than 500k. The reason is that most of the execution time comes from the overhead of Hadoop scheme such as disk I/O and communication costs. When the number of sequences is bigger than 500k, the total execution time increases linearly. We show the linear part of Figure 3(a) in more details in Figure 3(b). The linear equation of the regression line is y = 0.0005x + 1057.8, which means DPSP possesses very good scalability. Therefore, DPSP shows great practicability with large number of sequences. In the second experiment, we demonstrate the effect of increasing the length of POI. As shown in Figure 3(c), the total execution time goes up very quickly. The reason is that the number of candidate sequential patterns generated by each sequence grows exponentially as the length of POI increases. Therefore, the processing time of DPSP increases accordingly. The distributed nature of DPSP helps a little.

Finally, we show the advantages of the distributed scheme of our proposed algorithm DPSP. The datasets contain 1000k to 10000k sequences. As shown in Figure 3(d), the total execution time drops as the number of nodes increases from 1 to 8. This shows the merits of the distributed scheme. It is noted that both X-axis and Y-axis are in log scale. However, the overheads of disk I/O and message communication retard the reduction rate of the total execution time when the number of nodes equals to 13. Nevertheless, the decrease of the total execution time is remarkable. It is still worth to include more computing nodes in the cluster if we want to deal with more sequences. By utilizing Hadoop platform, it is extremely easy to extend the scale of the cluster to acquire better performance.

5 Conclusions

We proposed a distributed algorithm DPSP to address the inevitable scalability problem of the progressive sequential pattern mining. DPSP is running on top of Hadoop. We designed two Map/Reduce jobs in DPSP to efficiently compute candidate sequential patterns, update summaries of sequences, and assemble supports of candidate sequential patterns within each POI. As such, DPSP is able to report the most up-to-date sequential patterns. The experimental results show that DPSP possesses great scalability and thus increases practicability when the number of sequences become larger. In addition, by utilizing Hadoop platform, it is easy to increase the number of computing nodes in the cluster to acquire better performance.

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. of Intl. Conf. on Data Engineering, February 1995, pp. 3–14 (1995)
2. Aseervatham, S., Osmani, A., Viennet, E.: bitspade: A lattice-based sequential pattern mining algorithm using bitmap representation. In: Proc. of Intl. Conf. on Data Mining (2006)
3. Cheng, H., Tan, P.-N., Sticklen, J., Punch, W.F.: Recommendation via query centered random walk on k-partite graph. In: Proc. of Intl. Conf. on Data Mining, pp. 457–462 (2007)
4. Chilson, J., Ng, R., Wagner, A., Zamar, R.: Parallel computation of high dimensional robust correlation and covariance matrices. In: Proc. of Intl. Conf. on Knowledge Discovery and Data Mining, August 2004, pp. 533–538 (2004)
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified dataprocessing on large clusters. In: Symp. on Operating System Design and Implementation (2004)
6. Hadoop, `http://hadoop.apache.org`
7. Huang, J.-W., Tseng, C.-Y., Ou, J.-C., Chen, M.-S.: A general model for sequential pattern mining with a progressive database. IEEE Trans. on Knowledge and Data Engineering 20(9), 1153–1167 (2008)
8. Kargupta, H., Das, K., Liu, K.: Multi-party, privacy-preserving distributed data mining using a game theoretic framework. In: Proc. of European Conf. on Principles and Practice of Knowledge Discovery in Databases, pp. 523–531 (2007)
9. Luo, P., Xiong, H., Lu, K., Shi, Z.: Distributed classification in peer-to-peer networks. In: Proc. of Intl. Conf. on Knowledge Discovery and Data Mining, pp. 968–976 (2007)
10. Nguyen, S., Sun, X., Orlowska, M.: Improvements of incspan: Incremental mining of sequential patterns in large database. In: Ho, T.-B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS (LNAI), vol. 3518, pp. 442–451. Springer, Heidelberg (2005)
11. Wolff, R., Bhaduri, K., Kargupta, H.: A generic local algorithm for mining data streams in large distributed systems. IEEE Trans. on Knowledge and Data Engineering 21(4), 465–478 (2009)
12. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.J.: Scan: A structural clustering algorithm for networks. In: Proc. of Intl. Conf. on Knowledge Discovery and Data Mining, pp. 824–833 (2007)