An Efficient Map-Reduce Framework to Mine Periodic Frequent Patterns

A. Anirudh[1], R. Uday Kiran[2], P. Krishna Reddy[1], M. Toyoda[2], and M. Kitsuregawa[2,3]

[1] Kohli Center on Intelligent Systems, IIIT - Hyderabad, India
[2] Institute of Industrial Science, The University of Tokyo, Tokyo, Japan
[3] National Institute of Informatics, Tokyo, Japan
[1]alampally.anirudh@research.iiit.ac.in,[1]pkreddy@iiit.ac.in,
[2]{uday_rage, toyoda, kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. Periodic Frequent patterns (PFPs) are an important class of regularities that exist in a transactional database. In the literature, pattern growth-based approaches to mine PFPs have be proposed by considering a single machine. In this paper, we propose a Map-Reduce framework to mine PFPs by considering multiple machines. We have proposed a parallel algorithm by including the step of distributing transactional identifiers among the machines. Further, the notion of *partition summary* has been proposed to reduce the amount of data shuffled among the machines. Experiments on Apache Spark's distributed environment show that the proposed approach speeds up with the increase in number of machines and the notion of *partition summary* significantly reduces the amount of data shuffled among the machines.

Keywords: Data mining, Periodic frequent pattern mining, Map-Reduce

1 Introduction

Periodic frequent pattern (PFP) mining extracts regularities from transactional databases (TDB). A PFP is an itemset which is both frequent and periodic. For example, PFP mining extracts the knowledge about how regularly a set of items are being purchased by the customers from the super-market TDB. An example of PFP is $\{Bread, Butter\}$ $[support = 10\%, periodicity = 1\ hour]$. The preceding pattern demonstrates that both 'Bread' and 'Butter' are purchased in 10% of the transactions, and the maximum time interval between any two consecutive purchases containing both of these items is no more than an hour. The predictive behavior of PFPs could be used to improve the performance of several data mining based applications in the areas of customer relation management, inventory management, recommendation systems and so on.

In the literature, Tanbeer et al. [1] proposed a pattern-growth-based algorithm by considering a single machine. Several improvements to the approach proposed in [1] have been investigated [2–4] by considering a single machine. A Map-Reduced framework to exploit the power of thousands of machines is proposed in [5]. Encouraged by power of Map-Reduce paradigm, researchers are making efforts to propose parallel algorithms under Map-Reduce framework. A Map-Reduce based parallel FP growth approach to extract frequent patterns has been proposed in [6]. In this paper, we proposed a parallel PFP mining approach under Map-Reduce framework.

The PFP mining approach requires the processing of transaction identifiers (tids) for computing the periodicity. We have proposed a parallel approach which contains two Map-Reduce phases similar to [6]. In each phase the step to manage tids is integrated. Further, we have developed an improved approach by proposing the notion of *partition summary* in which instead of processing tid-list, the corresponding summary information is processed. Experimental results on three real-world datasets show that the proposed approach speeds up with the increase in number of machines and the notion of *partition summary* reduces the amount of data shuffled.

The rest of the paper is organized as follows. Section 2 discusses background. Proposed approach is discussed in Section 3. Performance evaluation is reported in Section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Mining periodic-frequent patterns on a single machine

Model of PFPs [1]: Let $I = \{i_1, i_2, \cdots, i_n\}$, $1 \leq n$, be a set of items. A set $X = \{i_j, \cdots, i_k\} \subseteq I$ is called a pattern. A transaction $t = (tid, Y)$ is a tuple, where tid represents a TID and Y is a pattern. A transactional database (TDB) over I is a set of transactions, i.e., $TDB = \{t_1, t_2, \cdots, t_m\}$, $m = |TDB|$. If $X \subseteq Y$, it is said that t contains X and such tid is denoted as tid_j^X. Let $TID^X = \{tid_j^X, \cdots, tid_k^X\}$, be the set of all tids where X occurs in TDB. The **support** of a pattern X is the number of transactions containing X in TDB, denoted as $Sup(X)$. Therefore, $Sup(X) = |TID^X|$. Let tid_i^X and tid_j^X be two consecutive tids where X appeared in TDB. The **period** of a pattern X is the number of transactions between tid_i^X and tid_j^X. Let $P^X = \{p_1^X, p_2^X, \cdots, p_r^X\}$, $r = Sup(X) + 1$, be the complete set of periods of X in TDB. The **periodicity** of a pattern X is the maximum difference between any two adjacent occurrences of X, denoted as $Per(X) = \max(p_1^X, p_2^X, \cdots, p_r^X)$. A pattern X is a PFP if $Sup(X) \geq minSup$ and $Per(X) \leq maxPer$, where $minSup$ and $maxPer$ represent the user-specified thresholds on minimum $support$ and maximum $periodicity$ respectively.

The existing periodic pattern growth (PF-growth) algorithm [1] accepts a TDB, $minSup$ and $maxPer$ as inputs and outputs a complete set of PFPs. The structure of periodic-frequent pattern tree (PF-tree) consists of a PF-list and a prefix tree. Here, the prefix tree of PF-tree explicitly maintains the tids for each occurrence of the pattern only at the tail-node of every branch unlike FP-tree which maintains support in every node. The algorithm consists of two database scans.

1) Finding one sized PFPs: In the first database scan, the PF-growth scans the entire database and discovers 1-patterns by computing support and periodicity values for each item. The final PF-list is generated after pruning the items that have failed to satisfy the $minSup$ and $maxPer$ constraints and sorting the items in the increasing order of support.

2) Construction of PF-tree: In the second database scan, the PF-tree is constructed by inserting the transactions according to PF-list order with tail-node carrying the tid. After completing two database scans PF-tree is constructed.

Mining of PFPs: The mining of PFPs start by constructing prefix tree (PT_i) for the last item i in the PF-list as an initial suffix item. For each item j in PT_i, all of its nodes' tid-list is aggregated to derive the tid-list of the pattern ij. If ij is a PFP, then j is considered to be periodic-frequent in PT_i. The conditional tree is constructed by choosing

Table 1: A running example of a Transactional Database

tid	Items	tid	Items	tid	Items	tid	Items	tid	Items
1	$bcdf$	4	$abde$	7	$acdef$	10	cd	13	de
2	$abdef$	5	e	8	abc	11	$bcde$	14	ae
3	b	6	bc	9	bf	12	abc	15	e

every periodic-frequent item j in PT_i, and is mined recursively to discover the patterns. After finding all PFPs for a suffix item i, it is pruned from the original PF-tree and the corresponding nodes' tid-lists are pushed to their parent nodes. The above steps are repeated until the PF-list becomes NULL.

2.2 Mining PFPs with Period Summary

The concept of *period summary* [4] was introduced to mine PFPs on a single machine, where only the summary information is stored instead of the tids in the tail-node. It has been shown that the notion of *period summary* results in lower memory consumption.

2.3 Map-Reduce framework

Map-Reduce [5] framework has been proposed to enable the processing of large datasets on a large cluster of commodity machines. Users specify the problem as a sequence of Map-Reduce steps. In each Map-Reduce step, the Map function processes key-value pairs and the reduce function merges all the values associated values with the same key.

2.4 Parallel FP-growth

Li et al. [6] proposed parallel FP-growth to extract frequent patterns using two Map-Reduce phases. The first phase constructs the F-list, which contains 1-sized frequent patterns. For each transaction, the mapper outputs key-value pairs as $\langle item, 1\rangle$ and reducer sums up all the ones for each item to count the corresponding support. The second phase constructs independent local FP-trees on different machines. For each transaction, all the sub-patterns are generated and mapper outputs the key-value pairs as $\langle$*partition-id,sub-pattern*$\rangle$ and reducer aggregates the transactions and construct local FP-trees. Frequent patterns are extracted at each worker by mining the local FP-trees.

3 Proposed Approaches

3.1 Parallel periodic frequent pattern growth (PPF-growth)

Both FP-growth (sec. 2.4) and PF-growth (sec. 2.1) mining are recursive pattern-growth approaches. The difference comes in the tree structure where additional tid information is processed which is required for computing the periodicity in PF-growth. The proposed approach consists of initial step, two Map-Reduce phases and the mining step.

1) Initialization: Initially, the TDB is segmented into multiple partitions. As an example, consider Table 1 as a TDB which is divided into two partitions with $minSup = 5$ and $maxPer = 4$. Here, 0^{th} and 1^{st} partitions contain transactions with tids between 1-8 and 9-15 respectively.

2) Map-Reduce phase 1 (first database scan): The phase is depicted in Fig. 1(a). In this step, parallel periodic frequent pattern list (PPF-list) is constructed by calculating

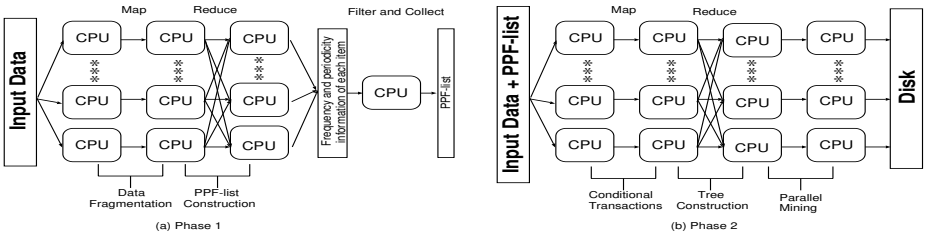

Fig. 1: Two phases of parallel periodic frequent pattern mining

Algorithm 1 PPF-listConstruction (TDB)

Procedure: Map($key = null, value = TDB_i$)
  for each transaction $t_{\text{cur}} \in TDB_i$ **do**
    for each item *it* in t_{cur} **do**
      Output (it, tid) // *tid* is the current transaction id

Procedure: Reduce($key = it, value =$ *TID-list*)
  Sort *TID-list* and initialize $id_l = 0$, $sup = 0$ and $per = 0$
  for each $tid \in$ *TID-list* **do**
    Set $sup+ = 1$, $per = max(per, tid - id_l)$ and $id_l = tid$

the support and periodicity values for each item. The Map-Reduce steps are as follows.
Map: For each item in a transaction, it outputs key-value pairs where key is the *item* and value is the *tid* of the current transaction ($\langle$*item, tid*$\rangle$).
Reduce: It groups all the tids of each item in a tid-list (Alg. 1). The tid-list is sorted and then its support and periodicity are computed (Fig. 2(a) - (c)).

The final PPF-list is obtained by filtering the items which do not satisfy the *minSup* and *maxPer* thresholds. After the master machine obtains the PPF-list, the items are sorted in decreasing order of their supports and are assigned a rank (to simplify the distribution of transactions using a hash function). The most frequent item is assigned a rank of 0, the second most frequent item is assigned a rank of 1 and so on.
3) Map-Reduce phase 2 (second database scan): The phase is depicted in Fig. 1(b). In this step, parallel periodic frequent pattern trees (PPF-trees) are constructed on each machine. The Map-Reduce steps are as follows.
Map: For each transaction, the items which are not present in PPF-list are filtered, translated into their ranks and are sorted in ascending order. Then all the sub-patterns (n sub-patterns will be generated) are extracted and are assigned to a partition based on a simple hash function $rank[item]\%numOfPartitions$. Here, *numOfPartitions* is the number of partitions available and *item* is the last item in a sub-pattern. The hash function gives a partition-id for which the pattern is responsible for further computation. Each sub-pattern is outputted as a key-value pair, with key as the *partition-id* and value as a tuple of *sub-pattern* and current *tid* ($\langle$*partition-id,(sub-pattern, tid)*$\rangle$).
Reduce: Independent local PPF-trees are constructed by inserting all the sub-patterns into the tree in the same order as the PPF-list with tid stored only in the tail-node of the branch. The process of tree construction (Alg. 2) is the same as the construction of

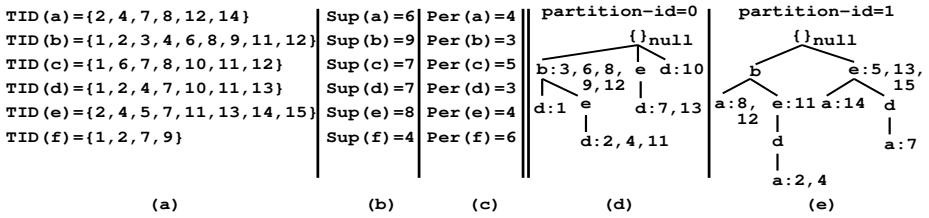

Fig. 2: Construction of PPF-list and prefix tree (a)-(c) Finding support and periodicity for each item (d) tree at partition-id 0 (e) tree at partition-id 1

PF-tree [1]. Trees constructed on two different machines are shown in Fig. 2(d), (e).

4) Mining of PPF-trees: Note that for any suffix item, the complete conditional tree information is available in the corresponding machine. So, during conditional pattern building, communication is not required between the machines and PFPs can be mined in parallel. Parallel mining of PFPs is similar to the mining process of PF-growth but worker machine processes only for those suffix items for which it is responsible for computation. This is checked by using the hash function $rank[it]\%numOfPartitions$. Here, *it* is the chosen suffix item. Mining for a suffix item (in increasing order of support) is done only if the output of hash function is equal to the partition-id of that machine. In the existing approach, the mining process of '*d*' occurs only after the mining process of '*a*' is completed. Whereas in the proposed approach, both the processes happen in parallel due to which the time taken to extract the PFPs is reduced.

3.2 PPF-growth using partition summary

A straight-forward approach for the first phase was discussed where large tid-lists are shuffled across machines. In [4], the notion of *period summary* is used to reduce the memory consumption by storing the interval information instead of tids in the tail-node of the PF-tree. Similarly, instead of tid-list for each item, only interval information can be processed in a partition. The summarized interval information concerning to an item for the given partition is called *partition summary*. We proposed an improved Map-Reduce based PF mining approach based on partition summary, which is defined as follows.

Definition 1. A **partition summary (PS)** captures the interval information of the item occurrences, the periodicity and support of respective item within that interval. That is, $PS = \langle tid_i, tid_j, per, sup \rangle$, where tid_i and tid_j, represents the first and last tids of that interval respectively, *per* is the periodicity and *sup* is the support of a pattern within the interval whose tids are within tid_i and tid_j.

The approach with PS is as follows: With PS, the phase 1 of PPF-growth proposed in the preceding section is modified. The map function is the same but instead of a reduce function, the *combine* function is applied which consists of three steps: initialization, intra-partition and inter-partition merging steps. During the initialization step, $PS = \langle 0,0,0,0 \rangle$ is initialized for each item. During the intra-partition merging step, the interval information in *PS* is updated for each item by iterating over its tids as explained

Algorithm 2 PPF-treeConstructionMining (TDB, PPF-list)

Procedure: Map($key = null, value = TDB_i$) // TDB_i is the segment of TDB

for each transaction $t_{\text{cur}} \in TDB_i$ **do**

filter and sort the elements in t_{cur} which are not in PPF-list

for $j = (|t_{cur}| - 1)$ to 0 **do**

partition-id = getPartition($t_{cur}[j]$)

if H does not contain partition-id **then**

Output(partition-id; ($t_{cur}[0:j]$, tid)) // tid is the current transaction id

Procedure: Reduce($key = partition\text{-}id, value = transactions$)

Initialize PPF-tree, T

for t_{cur} in transactions **do**

for *it* in t_{cur} **do**

if T does not have child *it* **then**

Create a new child node *it* and link it with the parent

Traverse to the child *it*

Add the current transaction id to the tid-list at the tail node of the transaction t_{cur}

Procedure: Map($key = partition\text{-}id, value = PPF\text{-}tree$) // Parallel Mining

for each suffix item *i* in PPF-list **do**

if current partition-id is responsible for item *i* **then**

Generate PT_i and CT_i and mine recursively in CT_i for patterns with suffix *i*

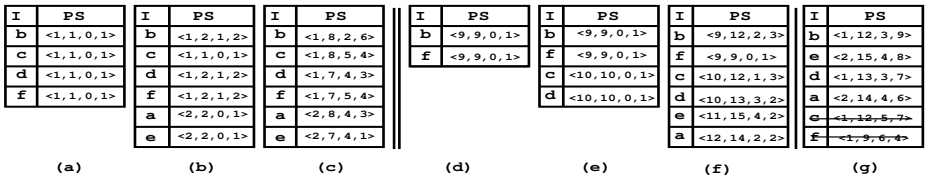

Fig. 3: Construction of PPF-list using Partition Summaries

in Alg. 3. All the *PS* for an item are merged into one PS during inter-partition merging step as explained in Alg. 3. Fig.3(a) - (c) shows the construction of PPF-list after scanning first transaction, second transaction and the entire shard of data assigned to partition 1. Similarly, Fig. 3(d) - (f) shows the construction of PPF-list in partition 2. Fig. 3(g) represents the final PPF-list constructed by merging the PPF-lists built on partition 1 and 2. The elements striked off are the ones which did not satisfy *minSup* and *maxPer* constraints.

4 Performance Evaluation

We have conducted the experiments to evaluate the speed up performance of the proposed approach. The algorithm is written in Python using Apache Spark architecture and the experiments are conducted on Amazon Elastic Map-Reduce (EMR) cluster with each machine of 8 GB memory. The runtime in the experiments specifies the total execution time of a Spark job. We employed 3 real-world datasets for conducting experiments, Retail store [7] (88,162 transactions with 16,470 items), Twitter dataset [2]

Algorithm 3 PPF-list with Partition Summaries

Procedure: Combine($key = it, value = TID\text{-}list$)

Initialize $PS = \langle 0,0,0,0 \rangle$ // (first tid, last tid, periodicity, support)
intra-partition: generatingSummaries (PS, $TID\text{-}list$)

$PS[0] = TID\text{-}list[0]$
for each tid $\in$ $TID\text{-}list$ **do**
$PS[2] = max(PS[2], tid - PS[1])$, $PS[1] = tid$ and $PS[3] += 1$
inter-partition: mergingSummaries (PS_1, PS_2, $\cdots$, PS_3)

Group all PS_i into $PS\text{-}List$
Sort $PS\text{-}List$ based on first tid
$Final\text{-}PS = <PS\text{-}List[0][0], 0, 0, 0>$
for each $PS \in PS\text{-}List$ **do**
$Final\text{-}PS[2] = max(Final\text{-}PS[2], PS[2], PS[0] - Final\text{-}PS[1])$
$Final\text{-}PS[1] = PS[1]$ and $Final\text{-}PS[3] += PS[3]$
return $Final\text{-}PS$

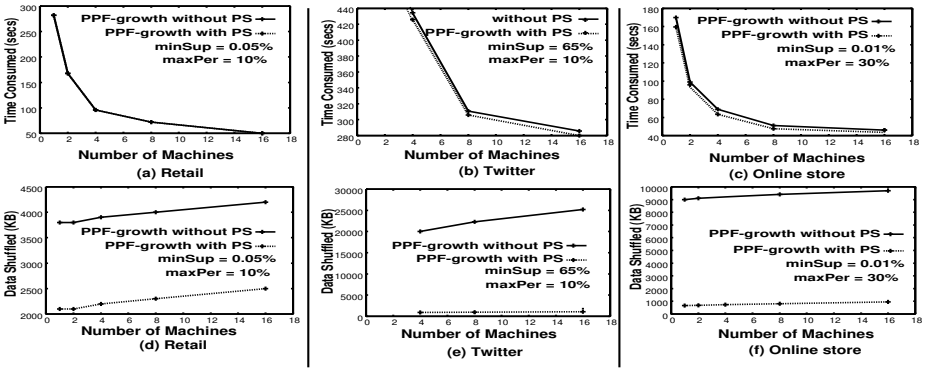

Fig. 4: Time Consumed and corresponding amount of data shuffled vs Number of Machines for different datasets at different *minSup* and *maxPer* values

(43,200 transactions with 44,201 items) and Online store [8] (541,909 transactions with 2,603 items).

The experiments are conducted by increasing the number of machines from 1 to 16 against the total time consumed. Fig. 4(a) shows variation of total time consumed with increase in number of machines for the Retail store dataset with $minSup = 0.05\%$ and $maxPer = 10\%$. As the number of machines is increased, the total time consumed decreases rapidly due to parallel computation. This shows that it is possible to improve performance with parallel computation. Here, the time taken for 1 machine is 282 seconds which is reduced to 72 seconds for 8 machines. The speedup can be computed as $(282/72)/8 = 48.95\%$. However, the algorithm reaches a saturation point for all the datsets and is obtained when 16 machines are used.

Fig. 4(b) shows the results for the twitter dataset by fixing $minSup = 65\%$ and $maxPer = 10\%$. It can be noticed it is not possible to extract patterns with one or two

machines as it is a dense dataset and the main memory is not sufficient to carry out recursive computation of patterns from PPF-tree. The Map-Reduce model enables the extraction of patterns by splitting the task among multiple machines and processing in parallel. Fig. 4(c) also shows a similar performance for Online store dataset.

Results with Partition Summary: Fig. 4(d) - (f) shows how the amount of data shuffled among the machines varies with and without using the notion of PS. It can be observed that there is a significant reduction of data shuffled with PS for all datasets. Fig. 4(a) - (c) shows the corresponding improvement in total time consumed with PS because of the reduced amount of data shuffled. Fig. 4(a) shows no improvement in time (overlapping lines) as it is a sparse dataset (Retail) and optimization using PS did not have much effect. Overall, the reason for minor improvement with PS is that in the experimental environment of Amazon EMR the data transfer speeds among the machines is very fast. So, the influence of reduction in data shuffled is not significant. However, we believe that in the slow LAN, and WAN environments, the effect of reduction in data shuffled will lead to significant improvement in total time consumed.

5 Conclusion

In this paper, we presented a parallel periodic frequent pattern extraction approach with Map-Reduce. Further, the notion of *partition summary* was introduced to reduce the amount of data shuffled among the machines. Experiments on massive datasets show that the proposed algorithm speeds up with the increase in number of machines. The modern e-Commerce applications and social network sites which normally collect huge datasets could exploit the proposed Map-Reduce based framework to extract knowledge of periodic-frequent patterns for improving the efficiency. As part of future work, we will develop algorithms for balanced load distribution for mining PFPs in parallel.

6 Acknowledgment

This research was partly supported by the program Research and Development on Real World Big Data Integration and Analysis of the Ministry of Education, Culture, Sports, Science and Technology, and RIKEN, Japan. We acknowledge K. Amulya for her contribution in implementation of the idea.

References

1. Tanbeer, S.K., Ahmed, C.F., Jeong, B.S., Lee, Y.K.: Discovering periodic-frequent patterns in transactional databases. In: PAKDD, Springer (2009) 242–253
2. Kiran, R.U., Shang, H., Toyoda, M., Kitsuregawa, M.: Discovering recurring patterns in time series. In: EDBT. (2015) 97–108
3. Amphawan, K., et al.: Mining periodic-frequent itemsets with approximate periodicity using interval transaction-ids list tree. In: WKDD. (2010) 245–248
4. Anirudh, A., Kiran, R.U., Reddy, P.K., Kitsuregawa, M.: Memory efficient mining of periodic-frequent patterns in transactional databases. In: IEEE, SSCI. 1–8
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1) (January 2008) 107–113
6. Li, H., et al.: Pfp: Parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM Conference on Recommender Systems. RecSys '08, ACM (2008) 107–114
7. Brijs, T., et al.: Using association rules for product assortment decisions: A case study. In: Knowledge Discovery and Data Mining. (1999) 254–260
8. Chen, D., et al.: Data mining for the online retail industry: A case study of rfm model-based customer segmentation using data mining. JDMCSM **19**(3) (2012) 197–208