2012 IEEE 12th International Conference on Data Mining Workshops

MapReduce-based Closed Frequent Itemset Mining with Efficient Redundancy Filtering

Su-Qi Wang*, Yu-Bin Yang*, Guang-Peng Chen*, Yang Gao* and Yao Zhang†
*State Key Laboratory for Novel Software Technology, Nanjing University Nanjing, China
†JinLing College, Nanjing University, Nanjing, China
Email: yangyubin@nju.edu.cn

***Abstract*—Mining closed frequent itemset(CFI) plays a fundamental role in many real-world data mining applications. However, memory requirement and computational cost have become the bottleneck of CFI mining algorithms, particularly when confronting with large scale datasets, which herewith makes mining closed frequent itemset from large scale datasets a significant and challenging issue. To address the above issue, a parallelized AFOPT-close algorithm is proposed and implemented in this paper based on the cloud computing framework MapReduce, which is widely used to cope with large scale data. Furthermore, an efficient parallelized method for checking if a frequent itemset is globally closed is also proposed on the MapReduce platform to further improve the mining performance. Experimental results are then provided and analyzed to verify the efficiency and effectiveness of the proposed methods for mining closed frequent itemset.**

***Keywords*-closed frequent itemset; MapReduce; data mining; AFOPT-close; Hadoop**

I. Introduction

The method for mining closed frequent itemset (CFI) was proposed in 1999 by Pasquier et al. [1] as an alternative of the traditional frequent itemset mining(FIM). An important advantage of CFI mining is that it has the same power as frequent itemset mining, but greatly reduces the number of redundant rules and increases both efficiency and effectiveness of mining. Since the introduction of CFI mining, it has been actively studied. The existing CFI mining algorithms can be classified into two categories: (1) candidate generate-and-test approach [1] and (2) pattern growth approach [2] [3] [4]. These algorithms have good performance when the size of dataset is small or the support threshold is high. However, when the scale of dataset grows large or the support threshold turns to be low, both memory use and communication cost are unacceptable. Some early efforts (the details are introduced in Section II) focused on speeding up the mining algorithms by running them on PC clusters. It may improve the mining performance, but also raises some other issues including workload balancing, data partitioning, global information developing from local nodes, minimization of communication costs, and potential errors caused by node's failure.

To overcome the above drawbacks, MapReduce was designed to support distributed computing in a so-called cloud computing paradigm, turning out to be an efficient platform for parallel data mining for larger scale datasets. In this paper, in order to accommodating the advantages brought by MapReduce for CFI mining, we design and implement a parallelized AFOPT-close [4] algorithm on Mapredue, which works in a divide-and-conquer way similar to the FP-growth algorithm, to mine the closed frequent itemset efficiently. Moreover, to further improve the mining performance, an efficient parallelized method for checking if a frequent itemset is globally closed is also proposed to filter out redundant frequent itemsets.

The main contributions of this paper can be summarized as follows:

- We propose a parallelized algorithm for mining closed frequent itemset on MapReduce platform.
- We present new definitions for locally CFI and globally CFI, based on which redundant itemsets can be filtered.
- We design a method for filtering the redundant itemsets and apply it to mine closed frequent itemsets on MapReduce.

The remainder of this paper is organized as follows. Section II presents some related work. In Section III, a parallelized AFOPT-close algorithm based on MapReduce is proposed, and a parallelized redundancy filtering method for checking if a frequent itemset is globally closed is also presented. Section IV illustrates the experimental results and comparisons with other similar algorithm to validate the performance of the proposed method. Finally, the conclusion remarks are provided in Section V.

II. Related Work

Research efforts [5] [6] have already been made to design the FP-growth algorithm capable of working across multiple threads under the memory-shared environment. But those approaches failed to break the bottleneck of the severely heavy memory requirement when facing large scale datasets. There are also some other research efforts on addressing more detailed issues such as communication cost, memory and I/O utilization [7] [8] [9]. For example, K.-Y. Whang et al proposed a method to parallelize execution of the FP-growth algorithm under shared-nothing environment [10].

978-0-7695-4925-5/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDMW.2012.24

They may achieve good scalability but still suffer from the same problem.

With the development of cloud computing, MapReduce platform was designed to enable the distributed processing of huge data on large computing clusters, with good scalability and robust fault tolerance. Thus, many improved frequent itemset mining algorithms based on MapReduce were then proposed. Examples are as follows. Haoyuan Li et al. proposed a parallelized FP-growth algorithm PFP based on MapReduce [11], which divided an entire mining task into some independent parallel sub-tasks, and achieved quasi-linear speedups. Besides scalability, PFP also enables us to design pattern growth approach based on MapReduce. In our previous work, the AFOPT-close algorithm has been discussed and implemented based on MapReduce [12] by using four steps to complete parallel AFOPT-close algorithm, three of which are MapReduce phases:

- **Step 1:** Parallel counting. Count the support of each item that appears in the DB (MapReduce pass).
- **Step 2:** Constructing the global F-List. Sort the items by their frequencies in descending order and exclude the items of which the support is lower than the minimal support value (denoted by ξ).
- **Step 3:** Parallel mining closed frequent itemset. Mine the locally closed frequent itemset in parallel (MapReduce pass).
- **Step 4:** Parallel filtering the redundant itemsets. Filter the frequent itemset which is locally closed but not globally closed (MapReduce pass).

TID	Transactions
1	f m g h a b
2	p c b a m f d
3	h m a f b
4	c b p a m f h
5	c b p f s r

Result of Step3
f p c b 3
m a b 3
f m a 4
f m 4
f p c 3
f p 3
f 5

Figure 1. An Example of the Original Transaction Database and the Mined CFIs

With the above four steps, we can mine the closed frequent itemsets correctly. Figure 1 illustrates a simple example, in which the left part is the original transaction database and the right part shows the mined CFIs by executing **Step 1**, **Step 2** and **Step 3** when $\xi = 3$. For the right part, the last item of each itemset is the support value. Obviously, there exists some redundant itemset which is closed in local but not in global, such as: {f m 4}, {f p c 3}, {f p 3}. The parallelized method in **Step 4** is then used to filter those redundant itemsets out. Figure 2 illustrates an example of it. First, each mapper reads one itemset from the mined CFIs, and outputs it n times, where n is the itemset's length and the keys are the items appeared in itemset. Second, each reducer collects the corresponding values and sorts the itemsets by their lengths in descending order in order to avoid superset checking, and then filters the redundant itemsets in parallel. Finally, the itemset whose key is the last item of the itemset is saved. This method works but also causes heavy communication costs and computation costs. Take the frequent itemset {f p c b 3} for example, where 3 is the support value of the itemset. The method needs to send this itemset for 4 times as :{f: f p c b 3}, {p: f p c b 3}, {c: f p c b 3}, and {b: f p c b 3}. Obviously, the above itemset will be sent to the reducers repeatedly for 4 times with different key values. If ξ is small enough, there may exist many long frequent itemsets which will be sent to the reducers repeatedly for many times. Therefore, the overall costs of this method are surprisingly high. In order to cope with this problem, we propose an efficient parallelized CFI mining algorithm which also incorporates a novel redundant itemset filtering process to decrease both communication costs and computation costs.

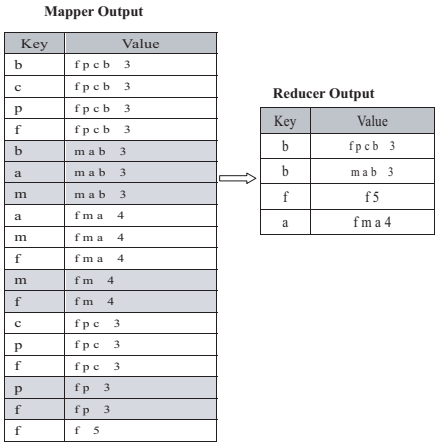

Figure 2. An Example of Redundant Itemset Filtering

III. An Efficient Redundant Itemset Filtering Approach

In this section, we present an efficient redundant itemsets filtering approach. First, we make new definitions in subsection III-A. Afterwards, we describe the proposed parallelized approach to filtering the redundant itemsets in subsection III-B.

A. Definitions

As mentioned above, mining the closed frequent itemsets in parallel based on MapReduce straightforwardly may lead

to the following problem: some itemsets may be closed in local but not globally. In this subsection, we provides clear definitions on *locally closed frequent itemsets*, and *globally closed frequent itemsets* respectively.

Definition III.1. ***(Locally Closed Frequent Itemsets).****The frequent itemset X is defined as* locally closed *if X is closed in its reducer in* ***Step 3****. The set L of such itemset as X is then called* Locally Closed Frequent Itemsets*.*

Definition III.2. ***(Globally Closed Frequent Itemsets).****The frequent itemset X is defined as* globally closed *if X is closed for all locally closed frequent itemsets. The set G of such itemset as X is then called* Globally Closed Frequent Itemsets*.*

Property. Suppose $X \in L$, if X is closed for all itemsets $\{Y | Y \in L \ and \ supp(X) = supp(Y)\}$, then $X \in G$.

Definition III.3. ***(Redundant Itemsets).****The frequent itemset X is defined as* redundant *if and only if $X \in L$, and $X \notin G$. The set R of such itemset as X is then called* Redundant Itemsets*.*

B. Redundant itemsets filtering

In our previous work [12], we proposed a basic method to filter redundant itemsets, which worked but brought heavy computation costs and communication cost to make itself time-consuming. In this paper, we propose a new method based on the definitions in subsection III-A to solve this problem. Naturally, we may implement an efficient redundant itemset filtering approach easily by selecting out the frequent itemset which is closed in global with the same support value. Therefore, we keep the support value as the key of an itemset, and the itemsets with the same support value will be sent to the same reducer. The pseudo-code of this method is described in Algorithm 1, in which the first three steps are the same to those in [12] (Suppose $X \in L$).

The processing flow of Algorithm 1 is described as follows. First, each mapper reads the result achieved from **Step 3** line by line and outputs a <key, value> pair:$< supp(X), X >$. Thus, the itemsets with the same support value will be sent to the same reducer and compressed into a tree. Second, the redundant itemsets are then filtered in parallel. For the example shown in Figure 1, we only need to send each itemset in locally closed frequent itemsets (the same as illustrated in the left part of Figure 1) for only once, while our previous method in [12] need to send each itemset at least 3 times. The detailed is depicted in Figure 3. Therefore, for a database containing n itemsets, the length of each itemset is $\{m_1, m_2, \cdots, m_n\}$. In [12], the itemsets need to be sent for $(m_1 + m_2 + \cdots + m_n)$ times; but for the proposed efficient filtering approach, we only need to send those itemsets for n times. That is to say, this method approximately reduces the communication costs for $(m_1 + m_2 + \cdots + m_n)/n$ times (that value is the same to the average length of the itemsets).

Algorithm 1 Efficient Redundant itemsets Filtering

```
1: Procedure:Map(key, value=supp(X)+X)
2: for each value do
3:    output(supp(X), X);
4: end for
5: end Procedure

6: Procedure:Reduce(key, Iterable values)
7: Define and initialize a tree: r;
8: Sort the itemsets by their lengths in descending order;
9: for each itemset in values do
10:   if itemset is closed in r then
11:      Insert the itemset into r;
12:   end if
13: end for
14: for each itemset in r do
15:   output(key, itemset);
16: end for
17: end Procedure
```

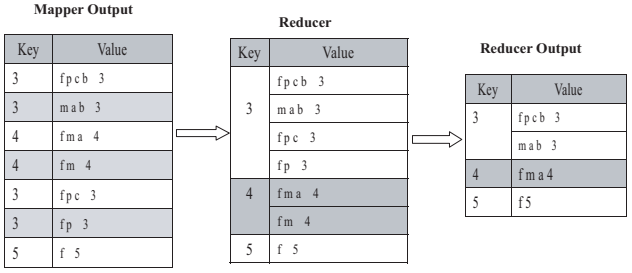

Figure 3. An example for efficient redundancy filtering

IV. Experimental Results

In order to verify the effectiveness and efficiency of the proposed method, we test it on two real datasets downloaded from [13]. The first one is "connect", which contains game state information with a size of 8.8 Megabytes; the second one is "webdocs", which contains exactly 1,692,082 transactions with 5,267,656 distinct items, in which the maximal length of a transaction is 71,472, and the size of the whole dataset is 1.4 Gegabytes.

Our experiments were all performed on a cluster with six nodes equipped with Hadoop 0.21.0 version, in which each node contains 4 Intel Core processors, 4GB RAM and 500G hard disk running Ubuntu 10.10. One node was designed as master, which was responsible for scheduling tasks' execution among different nodes; and other nodes were set as workers. Our algorithm was implemented using java and the JDK version is openjdk-6-jdk.

Figure 4 illustrates the experimental results tested on "webdocs". When ξ is set as $650,000$, our algorithm achieves the best speedup value. The reason is that, when ξ is larger, the local database for each reducer in **Step 3** is smaller. Thus, the time consumed by **Step 3** and **Step 4** is much shorter, while the time consumed by **Step 1** and **Step 2** almost keeps the same value.

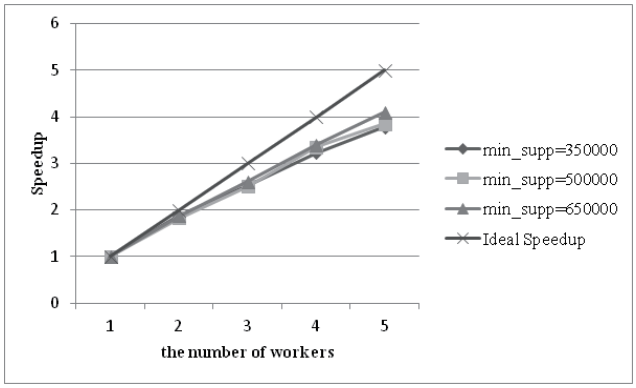

Figure 4. Experimental results on "webdocs"

Figure 5 shows the experimental results on "connect". Since the size of it is quite small, the speedup improvement is not so high as shown in Figure 4. As we can see from the figure, the speedup achieved by using five workers is less than that of four workers. The explanation is that the dataset is so small that the communication cost is much more than the computation cost for each node. The results indicate that our algorithm achieves good scalability for large scale datasets, but not for small ones.

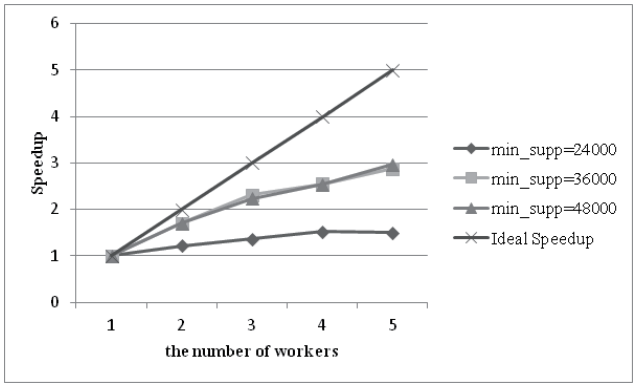

Figure 5. Experimental results on "connect"

We also make comparisons on the itemset sending times between our new redundancy filtering method and that in [12], as listed in Table I. Take the dataset "connect" for example, without the improved redundancy filtering method, if the itemset is too long, both computation cost and communication cost will be extremely expensive. From Table I, we can easily find that when $\xi = 24,000$, the itemset sending times in [12] is as 14.42 times as that in the proposed method, which validates the efficiency of the new filtering method clearly. However, the new method does not perform well on "webdocs", because the average length of the itemsets generated in **Step 3** is short. Therefore, experimental results demonstrate that our new algorithm is much more effective for long itemsets rather than short ones.

Table I
COMPARISONS ON ITEM SENDING TIMES

Dataset	ξ	Average length	sending times	
			our method	method in [12]
connect	24,000	14.43	317,321	4,577,943
	36,000	12.75	105,950	1,350,783
	48,000	10.84	33,274	360,685
webdocs	350,000	3.55	1,262	4,474
	500,000	3.04	164	468
	650,000	2.57	35	90

We also made running time comparisons between our new method and the method in [12], as listed in Table II. Experiments were carried out for the same two datasets with different threshold values on a cluster with five workers. It can be seen from Table II that the new method takes great advantages on the running time for "connect", because locally closed frequent itemset is larger than the original dataset and the itemset's average length is large as well. From the above analysis, the running speed of our new method is faster. However, as for "webdocs", our algorithm achieves nothing better when $\xi = 350,000$, $500,000$ and $650,000$. It is mainly because that the result generated in **Step 3** is too small. However, when $\xi = 200,000$, our algorithm is much faster because the result generated in **Step 3** is large and has much more long itemsets.

Table II
COMPARISONS ON RUNNING TIME

Dataset	ξ	Running time	
		our method	method in [12]
connect	24,000	416s	2,283s
	36,000	102s	230s
	48,000	91s	102s
webdocs	200,000	853s	unaccepted
	350,000	265s	266s
	500,000	206s	203s
	650,000	189s	177s

V. CONCLUSION

In this paper, we revisit the closed frequent itemset mining problem and propose a new method for filtering the frequent itemsets which are closed in local but not in global. Experimental results on two datasets show that our algorithm achieves good scalability on large-scale datasets. The results also reveal that when locally closed frequent itemset is large, communication cost becomes an important factor that decreases the performance of the algorithm, especially for

some mining tasks in which the threshold is very small or the itemset is long. The proposed method is able to solve this problem elegantly. In future, we will continue to improve the algorithm to make it more efficiently.

ACKNOWLEDGMENT

We would like to acknowledge the supports from the Program for New Century Excellent Talents of MOE China (Grant No. NCET-11-0213), National 973 Program of China (Grant No. 2010CB327903), the Natural Science Foundation of China (Grant Nos. 61273257,61035003, 61021062), the International Science and Technology Cooperation Program of China (Grant No. 2010DFA11030), and the Natural Science Foundation of Jiangsu, China (Grant Nos. BK2010054, BK2011005, BE2010638).

REFERENCES

[1] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," *Database Theory–ICDT'99*, pp. 398–416, 1999.

[2] J. Pei, J. Han, R. Mao *et al.*, "Closet: An efficient algorithm for mining frequent closed itemsets," in *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, no. 2, 2000, pp. 21–30.

[3] J. Wang, J. Han, and J. Pei, "Closet+: Searching for the best strategies for mining frequent closed itemsets," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 236–245.

[4] G. Liu, H. Lu, J. Yu, W. Wei, and X. Xiao, "Afopt: An efficient implementation of pattern growth approach," in *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

[5] O. Zaïane, M. El-Hajj, and P. Lu, "Fast parallel association rule mining without candidacy generation," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE, 2001, pp. 665–668.

[6] L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 1275–1285.

[7] M. El-Hajj and O. Zaiane, "Parallel leap: large-scale maximal pattern mining in a distributed environment," in *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, vol. 1. IEEE, 2006, p. 8.

[8] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz, "Toward terabyte pattern mining: an architecture-conscious solution," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 2–12.

[9] K. Chen, L. Zhang, S. Li, and W. Ke, "Research on association rules parallel algorithm based on fp-growth," *Information Computing and Applications*, pp. 249–256, 2011.

[10] I. Pramudiono and M. Kitsuregawa, "Parallel fp-growth on pc cluster," *Advances in Knowledge Discovery and Data Mining*, pp. 570–570, 2003.

[11] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*. ACM, 2008, pp. 107–114.

[12] G. Chen, Y. Yang, Y. Gao, and L. Shang, "Mining closed frequent itemset based on mapreduce," in *Proceedings of the 4th China Conference on Data Mining*. CCDM, 2011.

[13] http://fimi.cs.helsinki.fi/data.