An Efficient Algorithm of Frequent Itemsets Mining
Based on MapReduce*

Le Wang*, Lin Feng*, Jing Zhang, Pengyu Liao

School of Computer Science and Technology, Faculty of Electronic Information and Electrical Engineering, and School of Innovation and Experiment, Dalian University of Technology
Dalian 116024, China

Abstract

Mainstream parallel algorithms for mining frequent itemsets (patterns) were designed by implementing FP-Growth or Apriori algorithms on MapReduce (MR) framework. Existing MR FP-Growth algorithms can not distribute data equally among nodes, and MR Apriori algorithms utilize multiple map/reduce procedures and generate too many key-value pairs with value of 1; these disadvantages hinder their performance. This paper proposes an algorithm FIMMR: it firstly mines local frequent itemsets for each data chunk as candidates, applies prune strategies to the candidates, and then identifies global frequent itemsets from candidates. Experimental results show that the time efficiency of FIMMR outperforms PFP and SPC significantly; and under small minimum support threshold, FIMMR can achieve one order of magnitude improvement than the other two algorithms; meanwhile, the speedup of FIMMR is also satisfactory.

Keywords: Frequent Itemsets; Frequent Patterns; Big Data; MapReduce; Data Mining

1 Introduction

Frequent itemsets (patterns) mining is a topic in data mining, and algorithms are continually proposed such as [1]-[9] to address various issues arisen from this area. Among these algorithms, Apriori [9] is the first algorithm and it employs a level-wise approach; its weakness is that it requires multiple scans of dataset and generates candidate itemsets. FP-Growth is a classical pattern-growth approach and it overcomes the above weakness. While newer algorithms have significantly increased their performance, they are still not fast enough for dealing with large datasets. This situation gave rise to the development of parallel algorithms.

Along with the rapid growth of the application of MapReduce, more and more data mining algorithms have been ported to run on this framework, such as frequent pattern mining algorithms

*This work is supported by National Natural Science Foundation of P. R. China (No. 61173163, 51105052) and Liaoning Provincial Natural Science Foundation of China (No. 201102037).
*Corresponding author.
Email addresses: lelewater@gmail.com (Le Wang), fenglin@dlut.edu.cn (Lin Feng).

1548–7741 / Copyright © 2014 Binary Information Press
DOI: 10.12733/jics20103619
[14-19]. Just as their single threaded siblings, parallel algorithms for frequent itemset mining on MapReduce are based on either Apriori or FP-Growth. References [15, 17-20] are based on Apriori, and multiple Map/Reduce processes are needed; if the length of the longest frequent pattern is \( k \), \( k \) rounds of Map/Reduce must be applied. Algorithm PFP [16] is based on FP-Growth, and it needs only two rounds of Map/Reduce, but the data distributed to each data node are heavily redundant, and the size of data chunks cannot be well-proportioned, so its time-performance is still not very satisfying.

To address this issue, we propose a parallel frequent itemset mining algorithm that needs at most two rounds of Map/Reduce (in practice, only 1 round is needed in most circumstances), and distributes data evenly among nodes. Experimental results confirmed that our algorithm outperformed existing algorithms significantly in term of time efficiency.

The rest of this paper is organized as follows: Section 2 gives the problem definitions and related works; Section 3 proposes a new algorithm; Section 4 is the experiment results; and Section 5 is the conclusion.

2 Related Definition and Work

2.1 Related Definition

Let a dataset \( D = \{t_1, t_2, \cdots, t_n\} \) contains \( n \) transactions and \( m \) distinct items \( I = \{i_1, i_2, \cdots, i_m\} \). Each transaction \( T = (\text{TID}, P) \) in \( D \) consists of an itemset \( P \) and a unique identifier \( \text{TID} \) and \( P \subseteq I \). \(|D|\) represents the size of dataset \( D \). An itemset \( X = \{i_{j_1}, i_{j_2}, \cdots, i_{j_k}\} (1 \leq j_1 \leq j_2 \leq \cdots \leq j_k \leq m) \) containing \( k \) distinct items is called a \( k \)-itemset and \( k \) is the length of itemset \( X \).

**Definition 1** The minimum support threshold \( \text{minSup} \) is a user specified percentile of number of transactions in dataset \( D \); then minimum support number, \( \text{minSupNum} \), in \( D \) is defined by \( \text{minSupNum} = \text{minSup} \times |D| \).

**Definition 2** The support number (sn) of itemset \( X \) is the number of transactions containing \( X \) in a dataset.

**Definition 3** An itemset \( X \) is a frequent itemset if its support number is not less than the minimum support number.

**Definition 4** Let dataset \( D \) be divided into \( S \) little datasets \( D_1, D_2, \cdots, D_S \). \( \text{minSup} \times |D_i| \) is called the local minimum support number of \( D_i (1 \leq i \leq S) \). In dataset \( D_i \), itemset \( X \) is a local frequent itemset if its local support number is not less than the local minimum support number.

**Theorem 1** Any subset \( X (X \neq \text{null}) \) of a frequent itemset is a frequent itemset; any superset of a non-frequent itemset is not a frequent itemset [9].

2.2 Related Work

Frequent itemset mining algorithms can be categorized into two main types: the level-wise approach and the pattern-growth approach. The level-wise approach comes from Apriori [9] which
is the first algorithm for frequent itemset mining; it first generates frequent 1-itemsets, then using frequent 1-itemsets to generate frequent 2-itemsets by one scan of the dataset; apply the same approach to generate frequent k-itemsets \((k \geq 3)\) until no higher order frequent itemset can be found. Because this approach needs multiple scans of dataset, if the predefined minimum support threshold is small or the dataset is large, there could be too many candidate itemsets and the time & space performance of this algorithm is severely affected.

The pattern-growth approach was proposed in FP-Growth [7]. It utilizes iteration method, but not in the way of generating frequent \((k+1)\)-itemsets by combining frequent \(k\)-itemsets; it first finds all frequent 1-itemsets; then recursively discovers local frequent 1-itemsets on the condition of each frequent \(k\)-itemsets \((k \geq 1)\), to generate all frequent \((k + 1)\)-itemsets that contains this frequent \(k\)-itemset. FP-Growth compresses a dataset into an FP-Tree, and discovers local frequent 1-itemsets by searching this FP-Tree without scanning the dataset. FP-Tree is a compact tree structure; its construction process is as follows: (1) Sort the items in transaction itemset in descending order and delete non-frequent items; (2) Add the resulting itemset to the tree; itemsets with identical prefix share the same node.

PFP [16] is a MapReduce implementation of FP-Growth, and it needs two rounds of MapReduce. First, it creates a header table using MapReduce. Second, during the Map process, each transaction itemset is sorted in the order of the header table and non-frequent items are deleted from the itemset; for each item in the sorted itemset, the item itself is outputted as key and the itemset before this key item (sub transaction itemset) is outputted as value; that is, if a transaction itemset contains \(k\) items, it will produce \((k - 1)\) pairs \((key, value)\); the transaction itemset is distributed to multiple items in the header table to create sub trees. In the Reduce process, frequent itemsets are mined from corresponding sub transaction itemsets using FP-Growth. The main shortcoming of PFP is that it cannot distribute data uniformly among nodes, probably resulting much more data on some nodes than on others, and this hinders its overall performance.

Researches on MapReduce implementation of Apriori are abundant, such as those in [15, 17-20]. SPC [20] implements parallel Apriori by multiple rounds of MapReduce processes; if the largest length of frequent itemset is \(k\), \(k\) rounds of MapReduce is needed. The first round creates frequent 1-itemsets; then candidate frequent 2-itemsets are generated using combinations of the frequent 1-itemsets and are distributed to computing nodes; the second round of MapReduce scans the dataset counting the support number of each candidate on each node, and in Reduce process computing the total support number of each candidate on the whole dataset, so as to discover frequent 2-itemsets; in the same manner generates candidate \(k\)-itemsets, performs \(k\) rounds of MapReduce to scan the dataset and discover frequent \(k\)-itemsets \((k \geq 3)\). Paper [15] proposes an approximate Apriori algorithm using two rounds of MapReduce: in the first round, discovers local frequent itemsets on each node, and merges these itemsets to form the candidates; in the second round, scans the dataset for counting the support number of each candidate. The problem of this algorithm is that it outputs too many \((key, 1)\) pairs, and its mining result will lose some frequent itemsets.

## 3 Proposed Method FIMMR

FIMMR mines frequent itemsets by two rounds of MapReduce processes. In the first round, candidate itemsets are mined; in the second round, frequent itemsets are identified from the candidates.
Lemma 1 Let dataset $D$ be divided into $S$ equal-sized data chunks, denoted as $D = \{D_1, D_2, \cdots, D_S\}$; denote local frequent itemsets on these chunks as $FI_1, FI_2, \ldots, FI_S$ respectively, and the frequent itemset on dataset $D$ set $F1$. $F1$ is a subset of the union of all the chunks local frequent itemsets, that is, $FI \subseteq FI_1 \cup FI_2 \cup \cdots \cup FI_S$.

Proof Reduction to absurdity. Let $SN_i^X$ be the support number of frequent itemset $X$ on the $i$-th data chunk, then the support number of $X$ on the whole dataset $D$ is $\sum_{i=1}^{S} SN_i^X$. If $X$ is not a local frequent itemset of any data chunk, then the support number of $X$ on each data chunk is less than the local minimum support threshold, that is, $SN_i^X < minSup \ast |D_i|(1 \leq i \leq S)$, so $\sum_{i=1}^{S} SN_i^X < \sum_{i=1}^{S} minSup \ast |D_i| = minSup \ast \sum_{i=1}^{S} |D_i| = minSup \ast |D|$; this means that $X$ is not a frequent itemset, and this contradicts the assumption. So if an itemset is not one of the local frequent itemset, it must not be a frequent itemset.

According to Lemma 1, in the first round of MapReduce process, FIMMR mines local frequent itemsets using FP-Growth or Apriori algorithm as candidate itemsets.

Pruning strategy 1: If the sum of local support number of a candidate itemset on one or more data chunks is not less than the minimum support threshold, then it is identified as a frequent itemset, and removed from the candidate set to the frequent itemsets.

Lemma 2 Let $X$ be local frequent on data chunk $D_1$, $D_2$, $\cdots$, $D_j$, and its total support on these chunks is $S1$; let $X$ is not local frequent on data chunks $D_{j+1}$, $D_{j+2}$, $\cdots$, $D_S$, and its total support on these chunks is $S2$. If $S1 + \sum_{i=j+1}^{S} minSup \ast |D_i| - (S - j) < minSup \ast |D|$, then $X$ is not a frequent itemset.

Proof Because $X$ is not local frequent on data chunk $D_{j+1}$, $D_{j+2}$, $\cdots$, $D_S$, its support number on these chunks are less than the local minimum support threshold, that is: $S2 \leq (minSup \ast |D_{j+1}| - 1) + (minSup \ast |D_{j+2}| - 1) + \cdots + (minSup \ast |D_S| - 1) = \sum_{i=j+1}^{S} minSup \ast |D_i| - (S - j)$. If $S1 + \sum_{i=j+1}^{S} minSup \ast |D_i| - (S - j) < minSup \ast |D|$, then $S1 + S2$ must be less than the minimum support threshold, and $X$ is not a frequent itemset.

Pruning strategy 2: FIMMR would identify and delete some of the non-frequent itemsets using Lemma 2 to cut down the number of candidate itemsets.

Pruning strategy 3: Lemma 2 suggests that when lowering the local minimum support threshold, the maximum value of $S2$ in Lemma 2 will also be reduced. For example, taking the value of $\alpha \ast minSup(0 < \alpha \leq 1)$ as the local minimum support threshold, then $S2 \leq (\alpha \ast minSup \ast |D_{j+1}| - 1) + (\alpha \ast minSup \ast |D_{j+2}| - 1) + \cdots + (\alpha \ast minSup \ast |D_S| - 1) = \sum_{i=j+1}^{S} \alpha \ast minSup \ast |D_i| - (S - j)$. So as long as $S1 + \sum_{i=j+1}^{S} \alpha \ast minSup \ast |D_i| - (S - j) < minSup \ast |D|$, the aforementioned itemset $X$ in Lemma 2 is not a frequent itemset. FIMMR utilizes this property to reduce the number of candidate itemsets.

Pruning strategy 4: FIMMR also utilizes the closure property of frequent itemset to reduce the number of candidate itemsets (Theorem 1).

The pseudo code of the first round of MapReduce for mining candidate itemsets is illustrated in Fig. 1; the local mining process on each node is performed using either FP-Growth or Apriori.
When mining local frequent itemsets, taking a smaller value than the global minimum support threshold, $\alpha \ast \minSup$, as the local minimum support threshold. In Mapper phase, the frequent itemsets, their support and the size of the underlying data chunk is returned (see line 8 in Fig. 1). In Reduce phase, the total support of each frequent itemset and the total size of underlying data chunks, as well as the number of data chunks, are returned (line 15 in Fig. 1).

After the first round of MapReduce when candidate itemsets have been identified, they are further screened utilizing pruning strategy 1-4.

When finishing the pruning, if the set of candidates is not null, perform the second round of MapReduce to calculate the support of each candidate itemset. The pseudo code of the second round of MapReduce is as shown in Fig. 2.

```
SubProcedure IdentifyFrequentItemsets(D, min_sn, Cis)
Input: a global dataset D, the minimum support number min_sn, the global candidate itemsets Cis
Output: the frequent itemsets Fis
Method:
(1) Begin
(2) run(Mapper, Reducer)
(3) End
(4) Mapper(min_Sup*\alpha, data)
   //data is the local dataset assigned to the local map
(5) Begin
(6) Mining frequent itemsets Fis using FP-Growth or Apriori
(7) For each itemset inst in Fis
(8)     Output(inst, (count, [data]);
(9)     /\count is the support number of itemset inst
(10) /\datais the number of transactions in data
(11) EndFor
(12) End
(13)Reducer(<key, values>) /\key is a frequent itemset from map
(14)Begin
(15) Output(key, (sum(values.count), sum(values.data), count(values)))
(16) End
(18)Begin
(19) sum=0;
(20) For each value in values
(21)     sum=sum+value
(22) EndFor
(23) If sum>=min_sn
(24)     output(key, sum)
(25) EndIf
(26)End
```

Fig. 1: The first round of MapReduce of FIMMR

```
SubProcedure MiningCandidate(D, min_Sup*\alpha)
Input: a global dataset D, the minimum support threshold min_Sup
Output: the global candidate itemsets of data Cis
Method:
(1) Begin
(2) run(Mapper, Reducer)
(3) End
(4) Mapper(min_Sup*\alpha, data)
   //data is the local dataset assigned to the local map
(5) Begin
(6) Mining frequent itemsets Fis using FP-Growth or Apriori
(7) For each itemset inst in Fis
(8)     Output(inst, (count, [data]);
(9)     /\count is the support number of itemset inst
(10) /\datais the number of transactions in data
(11) EndFor
(12) End
```

Fig. 2: The second round of MapReduce of FIMMR

4 Experiment Results

In this section, we evaluate the performance of the proposed algorithm FIMMR and compare it with the algorithms PFP and SPC on two datasets: T20.I10.D10000K and T40.I20.D5000K. These three algorithms are precise algorithms, that is, they will get the same results under the same conditions; so our work is to compare the running time of these algorithms under various parameters. All algorithms were written in Python programming language. These two datasets were generated by the IBM data generator [9]. $T$ is the average length of transaction itemsets; $I$
is the average length of maximal potentially large itemsets; \( D \) is the total number of transactions. In algorithm FIMMR, the local minimum support is set \( 0.9 \times \minSup \).

The experimental platform is a cluster of 27 nodes, including 1 main node, 1 moderation node, and 24 data nodes. The hardware of each node is built on 2.5 GHZ dual-core CPU and 8 GB memory; software configuration is on Ubuntu 12.04 and Hadoop 0.22.0. Each node can carry out 2 Map processes simultaneously. To make full use the total 24 data nodes, each testing data file of this paper is cut to 24 pieces with equal sizes.

Fig. 3 shows the running time of the three algorithms FIMMR, PFP and SPC under different minimum threshold on two datasets. It is obvious that FIMMR outperforms the other two significantly. Because the smaller the minimum support threshold is, the more frequent itemsets there will be, so the running time will increase along with the decrease of the minimum support threshold. And because the number of frequent items increases quickly along with decrease of minimum support threshold, SPC generates too many candidates, and PFP generates too many long sub-transaction itemsets, so they both perform badly for small minimum threshold. On the other hand, FIMMR generates global candidate itemsets in the first round of MapReduce; and along with the pruning process, the number of candidates may reduce significantly, in many cases, the second round of MapReduce for counting support numbers of the candidates is no longer needed, so the performance of FIMMR is more stable.

Fig. 4 tests the time performance of the 3 algorithms on different size of datasets; the minimum support threshold is 1.0% and 1.5% in Fig. 4 (a) and Fig. 4 (b), respectively. When scale of data changes, the change of the number of frequent 1-itemsets is not quite significant, but the number of matching transactions for candidate itemsets increases proportionally in SPC, and the number of sub-transactions generated by PFP will also increase quickly, so basically the running time of SPC and PFP will increase proportionally to the size of the dataset. But as for the proposed algorithm, as shown in Fig. 4, its running time only increases slightly along the size of the dataset; and in many cases, because of the efficient pruning strategy, the second round of MapReduce for counting the support number of the candidates is skipped, leaving a more flat curve for FIMMR.

Fig. 5 is the speedup comparison of the three algorithms on different cluster size for two datasets. \( \text{Speedup} \) means the rate of performance enhanced when adding extra nodes to the cluster; that is, \( \text{speedup} = \frac{\text{old time}}{\text{new time}} \), where \( \text{old time} \) is the running time of an algorithm on one single node, and \( \text{new time} \) is the running time on multiple nodes. We can see that the speedup of

![Fig. 3: Running time under different minimum support threshold](image)

![Fig. 4: Speedup comparison of the three algorithms on different cluster size for two datasets](image)
both FIMMR and SPC are close to the Ideal speedup (linear); the reason they cannot be ideal is that when more nodes are added, communication cost between nodes will also increase, and this increases the overall running time. Algorithm PFP cannot distribute data equally among nodes, so its overall running time is significantly affected by the performance of the most heavily-loaded node. To summarize, the speedup of algorithm FIMMR and SPC are satisfactory.

5 Conclusion

We propose a frequent itemset mining algorithm FIMMR for the big data environment. FIMMR is built upon existing frequent itemset mining algorithms and the MapReduce framework; it first discovers parallelly local frequent itemsets as candidate; and the candidates are filtered; if the set of candidates is not null, the second round of MapReduce is performed to count the global support number of the remaining candidates. From the experimental results we can see that the proposed pruning strategy can filter the candidate itemsets efficiently, and in many cases, results null candidate itemset, rendering the second round of MapReduce unnecessary. The time efficiency of FIMMR outperforms PFP and SPC significantly, and under small minimum support threshold, FIMMR can achieve one order of magnitude improvement than the other two
algorithms; meanwhile, the speedup of FIMMR is also satisfactory.

References

[3] C. F. Ahmed et al., Efficient tree structures for high utility pattern mining in incremental databases, IEEE Transactions on Knowledge and Data Engineering, 21(12), 2009, 1708-1721
[5] D. Burdick et al., MAFIA: A maximal frequent itemset algorithm, IEEE Transactions on Knowledge and Data Engineering, 17(11), 2005, 1490-1504
[6] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, ACM SIGMOD International Conference on Management of Data, Dallas, TX, United States, 2000