International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 18 (2017) pp. 7355-7359
© Research India Publications. http://www.ripublication.com

Algorithms for Frequent Pattern Mining of Big Data

Syed Zubair Ahmad Shah[1], Mohammad Amjad[1], Ahmad Ali Habeeb[2], Mohd Huzaifa Faruqui[1] and Mudasir Shafi[3]

[1]Department of Computer Engineering, Jamia Millia Islamia, New Delhi, India.
[2]Department of Information Technology & Cloud, Ericsson India Global Services, Gurgaon, Haryana, India.
[3]Department of Electronics and Communication Engineering, Maharshi Dayanand University, Rohtak, Haryana, India.

[1]Orcid ID: 0000-0003-1847-8216

Abstract

Frequent pattern mining is a field of data mining aimed at unsheathing frequent patterns in data in order to deduce knowledge that may help in decision making. Numerous algorithms for frequent pattern mining have been developed during the last two decades most of which have been found to be non-scalable for Big Data. For Big Data some scalable distributed algorithms have also been proposed. In this paper, we analyze such algorithms in detail. We start with age old algorithms like Count Distribution and move on to traverse other algorithms one by one and finally to algorithms based on MapReduce programming model.

Keywords: Frequent Pattern Mining, Distributed Systems, Big Data Analytics

INTRODUCTION

Data mining is the task of scanning large datasets with the aim to generate new information or with the aim of knowledge discovery. Broadly speaking, data mining includes fields like clustering, frequent pattern mining (FPM), classification and outlier detection [1, 2, 3]. FPM is a popular data mining approach [4] which is used to find attributes that occur together frequently. Some of its important applications are market basket analysis, share market analysis, drug discovery, etc. [5]. Several FPM algorithms have been proposed in the last few decades such as Apriori [6], Eclat [7], FP-Growth [8], etc. [9, 10]. Identifying all frequent patterns is a complex task and is also computationally costly due to a large number of candidate patterns and so most algorithms become inefficient when it comes to the kind of data we deal with in this age, the Big Data. Social networking places like Facebook produce over 300TB of data every single day [11]. Amazon, Walmart and other giants register billions of transactions every year. Gene sequencing platforms can generate terabytes of data. To cope up with the problem of huge volumes of data, researchers proposed several parallel algorithms for the FPM problem.

This paper reviews some of the classical as well as recent parallel FPM algorithms. Various parallel FPM algorithms have been discussed and analyzed in detail starting with the classical algorithms and then moving onto the state of the art algorithms.

ALGORITHMS FOR PERFORMING FREQUENT PATTERN MINING ON A DISTRIBUTED ENVIRONMENT

A number of algorithms have been proposed for efficient frequent pattern mining on a distributed system which include Count Distribution, Data Distribution and Candidate Distribution algorithms by Agrawal and Shafer [12], Parallel Eclat by Zaki el. al. [13], Parallel FP-growth algorithm by Li el. al. [14], Single Pass Counting, Fixed Passes Combined-counting and Dynamic Passes Combined-counting algorithms by Lin et. al. [15] and Distributed Eclat algorithm by Moens et. al. [16]. We discuss a few of these herein.

Count Distribution Algorithm

Notations:

k-itemset	Itemset with k items
$Freq_k$	Set of frequent k-itemsets
$Cand_k$	Candidate itemset of size k
$Proc^i$	Processor i
DS^i	Dataset local to $Proc^i$
DSR^i	Dataset local to $Proc^i$ after re-partition
$Cand_k^i$	Candidate set of $Proc^i$ during pass k

The Count Distribution (CD) algorithm tries to minimize interaction. It partitions the dataset in horizontal manner into equal parts for the various processors. Each processor autonomously runs the Apriori algorithm and creates the hash-tree on the locally stored transactions. After each iteration, global count of candidates is calculated by adding all counts

over all processors. This algorithm has a limitation that every processor has to keep the whole hash-tree in its memory. Data distribution algorithm has to be used if the whole hash-tree cannot be stored in the memory. This algorithm permits superfluous calculations in parallel to prevent interaction.

In the first pass, every processor produces its local candidate itemset $Cand_1^i$. For remaining passes, the algorithm is given below:

1. Every processor Proc^i produces candidate set $Cand_k$ by making use of the frequent itemset $Freq_{k-1}$ that was produced in the previous iteration.
2. $Proc^i$ performs a pass on its DS^i and to find local support counts for candidates in $Cand_k$.
3. $Proc^i$ synchronizes with all other processors and broadcasts local $Cand_k$ counts to develop global $Cand_k$ counts.
4. Every processor $Proc^i$ calculates $Freq_k$ from $Cand_k$ using the global support count.
5. If $Freq_k$ is a null set then the algorithm terminates, or else it continues. Every processor $Proc^i$ will independently make this decision.

Figure 1 shows the speedup of CD algorithm (on datasets D1456K.T15.14 and D1140K.T20.16) as the number of processors is increased.

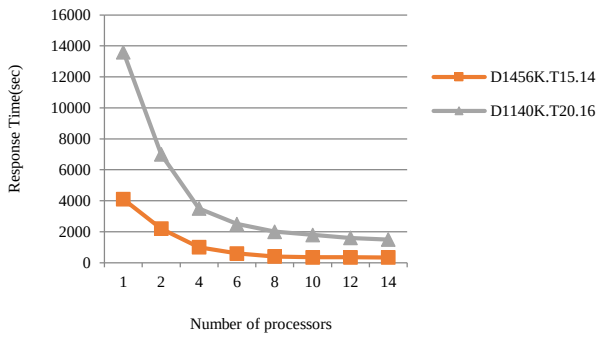

Figure 1: Speedup of Count Distribution Algorithm with increase in the number of processors

Data Distribution Algorithm

This algorithm uses memory of the system extra efficiently. The itemsets are partitioned and shared with processors. Each processor must find the count of its local subset of candidate itemsets and broadcast local data to remaining processors. Partitioning and the distribution of itemsets may affect performance of the algorithm, so it is desirable to have balanced load.

Pass 1:

This step is same as the step for k=1 in CD algorithm.

Pass k>1:

1. Processor $Proc_i$ produces candidate itemset $Cand_k$ from $Freq_{k-1}$ but keeps only a subset $Cand_k^i$ which has only $\frac{1}{N}$ itemsets of $Cand_k$. Union of all subsets generated by every processor is $Cand_k$ and their intersection is a $\emptyset$.
2. $Proc^i$ uses both local data and data received from remaining processors to develop support counts for the itemsets in its $Cand_k^i$.
3. Processor $Proc^i$ calculates $Cand_k^i$ using the local candidate set $Cand_k^i$ after getting complete support count data. Again, the intersection of all $Cand_k^i$ sets is $\emptyset$ and their union is $Freq_k$.
4. All processors synchronize and exchange their local $Cand_k^i$ with other processors. Every processor has the complete $Freq_k$ and can decide whether to continue the algorithm or to terminate it independently.

Processors find the support count for local itemsets in step 2 and broadcast or receive local data. Network congestion may become a problem in this setup so steps must be taken to prevent that.

Parallel Eclat

In Parallel Eclat (Par-Eclat), data is in vertical layout which is done by transforming the horizontal data into vertical itemset list. The two phases in the Parallel Eclat are given below:

Initialization Phase:

There are 3 sub-steps in the initialization step:

1. The support count for 2-itemsets are read after the preprocessing step has been done and all the frequent itemsets having count>= minimum support are introduced into $Freq_2$.

   ($Freq_k$ is the set of frequent k-itemsets).
2. Any of the clustering schemes between these two - the equivalence class or maximal hypergraph clique clustering - is applied to $Freq_2$ to produce the set of possible maximal frequent itemsets which are then partitioned among all the processors to achieve load-balancing.
3. The database is repartitioned so that every processor obtains the tid-lists of all the 1-itemsets in the cluster appointed to it.

Asynchronous Phase:

After the initialization phase, the tid-lists are accessible on local storage on every processor, so every processor can use

its assigned maximal cluster to generate frequent itemsets without synchronizing with any other processor. A cluster is processed completely before the processor moves onto the subsequent cluster. Local database is examined one time to complete this step. Therefore, there is less I/O overhead. Since a sublattice is induced by each cluster, either bottom-up traversal is used to produce all frequent itemsets or hybrid traversal is used to produce the maximal frequent itemsets, depending on the algorithm. Initially only tid-list for 1-itemset are available locally using which, the tid-lists for 2-itemsets clusters can be generated. These clusters are generally not big and so keeping the emerging tid-lists in memory creates no problem. 3-itemsets are produced by intersection of the tid-lists for 2-itemsets in the bottom-up approach. If the cardinality of the emerging tid-list >= minimum support, the new itemset is added to $Freq_3$. Then the resulting frequent 3-itemsets, $Freq_3$ are divided into sets of equivalence classes on the basis of common pre-fixes of size 2. Intersection of all pairs of 3-itemsets within an equivalence class is used to determine $Freq_4$. The process can be repeated till all frequent itemsets are found. After finding $Freq_k$ $Freq_{k-1}$ is deleted. Memory space is thus needed for the itemsets in $Freq_{k-1}$ only within one maximal cluster. This algorithm is thus main memory space efficient. For top-down stage of the hybrid-traversal, memory needs to keep track of only the maximal element seen so far, along with the itemsets not yet seen.

Par-Eclat Algorithm

1. Create $Freq_2$ using 2-itemset support counts
2. Produce clusters from $Freq_2$ using Equivalence Classes
3. Assign clusters to processors
4. Scan local dataset part
5. Share appropriate tid-lists with remaining processors
6. Procure tid-lists from remaining processors
7. *foreach* assigned cluster *C* find frequent itemsets using Bottom-Up traversal or Hybrid traversal

Figure 2 compares the execution time of Par-Eclat algorithm with CD algorithm [12] (for T10.I4.D2084K dataset). Par-Eclat clearly outperforms CD algorithm.

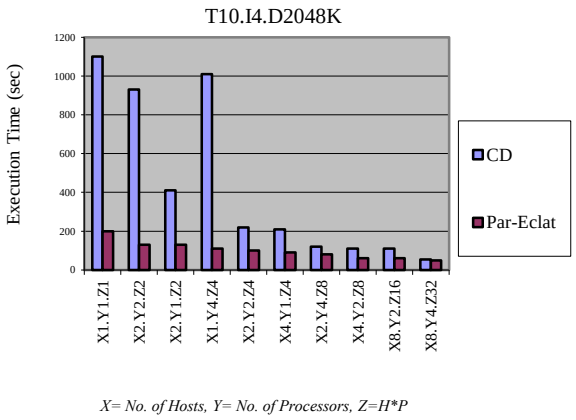

Figure 2: Comparison of Execution Time of Par-Eclat and Count Distribution

Single Pass Counting

It is the parallel implementation of Apriori algorithm using MapReduce. It makes Apriori efficient for mining association rules from massive datasets.

The main step in Apriori algorithm is finding the frequency of candidate itemsets. This process of counting can be parallelized. In each step, the mapper program emits <x, 1> for every candidate x that is there in the transaction. The reducer collects the key-value pair emitted by the mapper and sums the value for each key, which gives the total frequency of the candidate in the database. The reducer outputs all the candidates which have enough support count. Single Pass Counting (SPC), at *k*-th pass of database scanning in a MapReduce phase, finds out the frequent *k*-itemsets.

Map(key, value = itemset in transaction t_i):

Input : Dataset D_i

foreach transaction $tr_i \in D_i$ do

foreach item j $\in$ tr_i do

emit <j, 1>

end

end

Reduce(key=item, value = count):

foreach key k do

foreach value val in k do

k.count+=val

end

if k.count>=min. supp. count

emit <k, k.count>

end

end

Phase-1 of SPC algorithm

In phase-2, the algorithm generates frequent 2-itemset using frequent itemsets $Freq_1$ which are located at the Distributed Cache in Hadoop.

Map(key, value = itemset in transaction t_i):

Input : Dataset D_i and $Freq_{k-1}$(k>=2).

Fetch $Freq_{k-1}$ from *DistributedCache.*

Build a hash tree for $Cand_k$=gen-apriori($Freq_{k-1}$).

foreach transaction $tr_i \in D_i$ do

$Cand_t$ = subset($Cand_k$, tr_i)

foreach item i $\in Cand_t$ do

emit <i, 1>

end

end

Reduce(key=item, value = count):

foreach key *k* do

foreach value *val* in *k* do

k.count+=*val*

end

if k.count>=minimum_support_count

emit <*k*, *k*.count>

end

end

Phase-2 of SPC algorithm

Algorithm SPC:

1. Run Phase- 1 to find $Freq_1$
2. Run Phase-2 to find $Freq_2$
3. forall *k*>2 and $Freq_{k-1} \neq \phi$

Map function of phase-2

Reduce function of phase-2

end

Distributed Eclat

Distributed Eclat (Dist-Eclat) algorithm is based on Eclat algorithm for mining frequent itemsets from large datasets in parallel using MapReduce. This algorithm focuses on speed. Dist-Eclat is quite fast but it requires huge main memory as it stores the complete dataset in memory in vertical format.

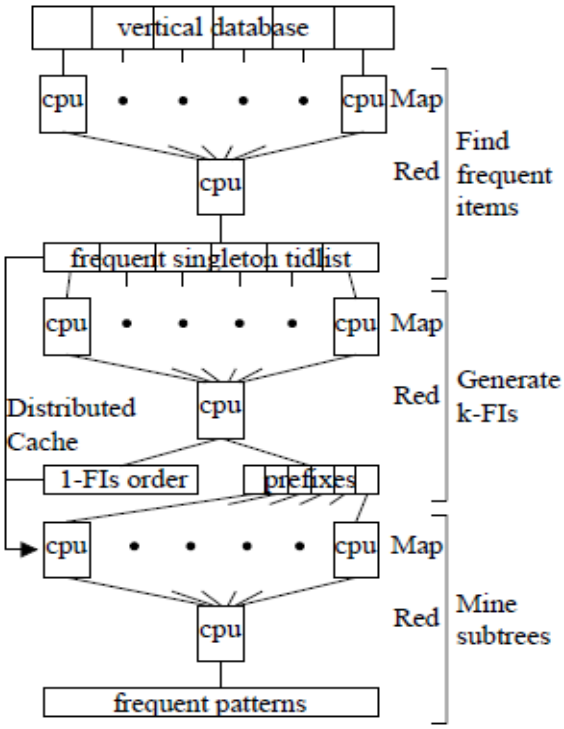

Figure 3: Eclat on MapReduce Framework

Dist-Eclat is 3-step method (figure 3). Each step can be distributed between mappers to maximise efficiency.

1. *Finding the Frequent Items*: In this step, the vertical database is partitioned into equal-sized sub-databases called shards and distributed among several mappers. Mappers extract the frequent singletons from their shard and send them to reducer. All frequent items are gathered in the reduce phase.
2. *k-Frequent Itemsets Generation*: The set of frequent k-itemsets, $Freq_k$, is generated. First, each mapper is assigned the combined form of local frequent item sets. Mappers find the frequent supersets of size k of the items using Eclat algorithm and running it up to level k. Then a reducer assigns the frequent itemsets to a new set of mappers. Distribution to mappers is performed by employing Round-Robin algorithm.
3. *Subtree Mining*: This step uses Eclat algorithm to mine the prefix tree from the assigned subsets. Subtrees can be mined independently by mappers since sub-trees do not need information from other sub-trees.

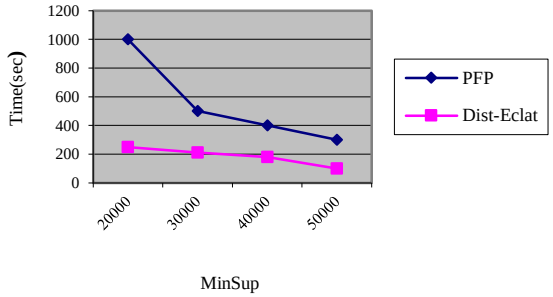

Figure 4: Comparison of Computation Time of Dist-Eclat and PFP

Figure 4 compares the runtime of Dist-Eclat with Parallel FP-Growth (PFP) [14] over del.icio.us dataset [17]. Minimum support threshold is denoted by the x-axis and the y-axis denotes the time taken by computation in seconds. Dist-Eclat is faster of the two algorithms.

CONCLUSION

In this paper, we traversed through many distributed frequent pattern mining algorithms in order to examine them and to present a picture of how these algorithms grew over time. We also presented a comparison of their runtimes on different datasets. We presented both types of algorithms – those based on message passing and those based on MapReduce approaches.

REFERENCES

[1] Han, J., Pei, J., and Kamber, M., 2011, Data Mining: Concepts and Techniques, Morgan Kaufmann Publishers, California, USA.

[2] Shah, S. Z. A., and Amjad, M., 2016, Classical and Re-learning based Clustering Algorithms for Huge Data Warehouses, Proc. 2nd International Conference on Computational Intelligence & Communication Technology, Ghaziabad, pp. 209-213.

[3] Aggarwal, C. C., 2015, Data Mining, Springer International Publishing, Switzerland.

[4] Aggarwal, C. C., and Han, J., 2014, Frequent Pattern Mining, Springer International Publishing, Switzerland.

[5] Berry, M. J. A., and Linoff, G. S., 2004, Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management, John Wiley & Sons Inc., Indiana, USA.

[6] Agrawal, R., and Srikant, R., 1994, Fast Algorithms for Mining Association Rules, Proc. 20th International Conference on Very Large Data Bases, Santiago, Chile, 1215, pp. 487-499.

[7] Han, J., Pei, J., and Yin, Y., 2000, Mining Frequent Patterns Without Candidate Generation, Proc. ACM SIGMOD International Conference on Management of Data, Dallas, Texas, pp. 1-12.

[8] Zaki, M. J., 2000, Scalable Algorithms for Association Mining, IEEE Transactions on Knowledge and Data Engineering, 12(3), pp. 372-390.

[9] Sucahyo, Y. G., and Gopalan, R. P, 2004, CT-PRO: A Bottom Up Non Recursive Frequent Itemset Mining Algorithm using Compressed FP-Tree Data Structures, Proc. ICDM Workshop on Frequent Itemset Mining Implementations, Brighton.

[10] Uno, T., Kiyomi, M., and Arimura, H., 2004, Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets, Proc. ICDM Workshop on Frequent Itemset Mining Implementations, Brighton.

[11] Lin, J., and Ryaboy, D., 2013, Scaling Big Data Mining Infrastructure: The Twitter Experience, ACM SIGKDD Explorations Newsletter, 14(2), pp. 6-19.

[12] Agrawal, R., and Shafer, J., 1996, Parallel Mining of Association Rules, IEEE Transactions on Knowledge and Data Engineering, 8(6), pp. 962- 969.

[13] Zaki, M. J., Parthasarathy, S., Ogihara, M., and Li, W., 1997, Parallel Algorithms for Discovery of Association Rules, Data Mining and Knowledge Discovery, 1(4), pp. 343-373.

[14] Li, H., Wang, Y., Zhang, D., Zhang, M., and Chang, E. Y., 2008, Pfp: Parallel FP-growth for Query Recommendation, Proc. ACM Conference on Recommender Systems, Lausanne, pp. 107-114.

[15] Lin, M. Y., Lee, P. Y., and Hsueh, S. C., 2012, Apriori-based Frequent Itemset Mining Algorithms on MapReduce, Proc. 6th International Conference on Ubiquitous Information Management and Communication, Kuala Lumpur, Article no. 76.

[16] Moens, S., Aksehirli, E., and Goethals, B., 2013, Frequent Itemset Mining for Big Data, Proc. International Conference on Big Data, Santa Clara, California, pp. 111-118.

[17] Wetzker, R., Zimmermann C., and Bauckhage, C., 2008, Analyzing Social Bookmarking Systems: A del. icio. us Cookbook, Proc. Mining Social Data Workshop, Patras, pp. 26-30.