ISSN 2395-1621

Frequent Itemsets Parallel Mining Algorithms

[#1] Suraj Ghadge, [#2]Pravin Durge, [#3]Vishal Bhosale,[#4]Sumit Mishra

[1]Surajghadge2908@gmail.com
[2]Pravindurge1906@gmail.com
[3]Vishalbhosale2012@gmail.com
[4]Mishra.sumit1994@gmail.com

[#1234] Department of Computer Engineering, JSPM's ICOER

ABSTRACT

This paper gives an overview of various parallel mining algorithms used for mining frequent itemsets in a database. We summarize these parallel mining algorithms on the basis of I/O overhead, data distribution, storage, scalability, load balancing, automatic parallelization and fault tolerance. On the basis of comparisons done, we get the most efficient parallel frequent itemsets mining algorithm i.e. FiDoop using FIUT and MapReduce programming model. Compared to related work, FIUT has four major advantages. First, it minimizes I/O overhead by scanning the database only twice. Second, the FIU-tree is an improved way to partition a database, which results from clustering transactions, and significantly reduces the search space. Third, only frequent items in each transaction are inserted as nodes into the FIU-tree for compressed storage. Finally, all frequent itemsets are generated by checking the leaves of each FIU-tree, without traversing the tree recursively, which significantly reduces computing time.

ARTICLE INFO

Article History

Received :18th Septmber 2015

Received in revised form :19th Septmber

Accepted : 2nd Septmber, 2015

Published online :

26th Septmber 2015

Keywords— **Frequent Itemsets, Parallel Mining Algorithms, MapReduce, Hadoop cluster,Ultrametric Trees**

I. INTRODUCTION

Data mining is a process of discovering pre previously unknown and useful information from large databases. The most widely used data mining technologies include association rules discovery, clustering, classification, and sequential pattern mining. Among them, the most popular technology is association rules discovery, which is mining the possibility of simultaneous occurrence of items, and then building relationships among them in databases.

Association rules mining can be divided into two parts: find all frequent itemsets, and generate reliable association rules straightforward from all frequent itemsets. Because frequent itemsets mining is the most time-consuming procedure, it plays an essential role in mining association rules. The algorithms developed for mining frequent itemsets can be classified into two types [1], [2]: the first is the candidate itemsets generation approach, such as Apriori algorithm, called Apriori-like; another aspect is a method without candidate itemsets-generation approach, such as FP-growth algorithm, called FP-growth-like.

Frequent itemsets mining (FIM) is a core problem in association rule mining (ARM). Speeding up the process of FIM is critical and indispensable, because FIM consumption accounts for a significant portion of mining time due to its high computation and input/output (I/O) intensity. When datasets in modern data mining applications become excessively large, sequential FIM algorithms running on a single machine suffer from performance deterioration. To address this issue, various parallel mining algorithms were proposed and implemented. In this paper we summarize various parallel mining algorithms and conclude with the most efficient of them all.

The remainder of this paper is organised as follows. Section II describes the various Apriori-like parallel mining

algorithms. Section III describes FP-Growth like parallel mining algorithm. Section IV gives an overview of the process of FiDoop on MapReduce. Section V discusses the related work. Finally, Section VI concludes this paper.

II. APRIORI-LIKE PARALLEL ALGORITHMS

In this section,we gave an overview of various Apriori like parallel algorithms.We have also mentioned about the demerits of these algorithms.

A. Apriori-based frequent itemset mining algorithms on MapReduce[3]

Three algorithms were proposed named SPC, FPC, and DPC to investigate effective implementations of the Apriori algorithm in the MapReduce framework.

1. SPC is a simple conversion of the serial Apriori algorithm into the distributed MapReduce version.
2. Example: A database of six transactions, min_sup = 33%, three mappers, and two reducers.In phase-1, the mapper handles transaction t1 by outputting <A, 1>, <B, 1>, and<C, 1> pairs and handle the transaction t2 by outputing<A, 1> and<C, 1>pairs.The combine function then is invoked to sum up the counts, and outputs <A, 2>, <B, 1>, and <C, 2> finally. The reducer sums up the counts associated with each key (i.e. item-id) and outputs items having counts at least 2 into L1.

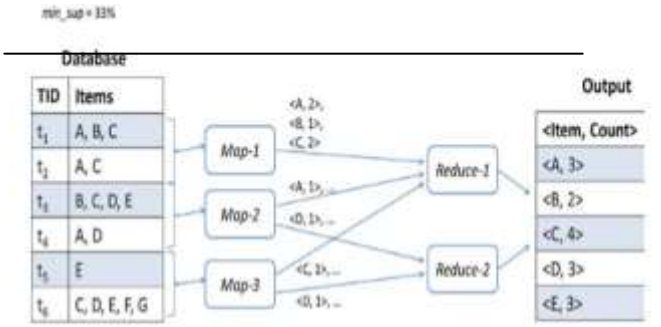
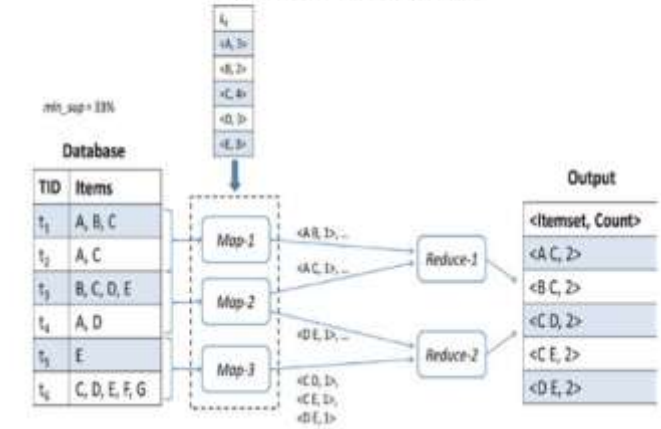


Figure 4. An execution example of Phase-1.



SPC ALGORITHM

1. phase 1 – find L1
2. phase 2 – find L2
3. for(k=3;Lk-1=fy;k++)
4. Map function
5. Reduce function
6. end.

3. FPC: FPC improves SPC by using a mapper to count the candidate.FPC combines candidates from several phases in SPC and performs the support counting in a single map-reduce phase.

FPC ALGORITHM

1. phase 1 – find L1
2. phase 2 – find L2
3. for(k=3;Lk-1=fy;k+=3)
4. Map function
5. Reduce function
6. end.

4. DPC: DPC dynamically combines candidates of consecutive passes by considering the workload of workers. DPC is proposed to strike a balance between reducing the number of map-reduce phases (by combining variable-length candidates) and increasing the number of pruned candidates

DPC ALGORITHM

1. phase 1 – find L1
2. phase 2 – find L2
3. for(k=3;Lk-1=fy)
4. Map function
5. Reduce function
6. k+=(counter+1)
7. end.

Demerits- FPC & DPC might suffer from overloading candidates for a mapper if the number of candidates after merging is too large. Cost of database loading in the beginning of each phase is relatively high if only few candidates are counted in a phase.

B. Parallel Mining of association rules using Apriori ˆ lgorithm[4]

Discovering association rules is an important data mining problem. Recently there has been considerable research in designing fast algorithm for this task. Since databases to be mined are often very large so parallel algorithms are required.In order to determine the best rules for mining in parallel, we explore a spectrum of trade-offs between computation , communication, memory usage ,synchronization and the use of problem-specific in parallel data mining.

Specifically,

1) The focus of the Count Distribution algorithm is on minimizing communication. It does so even at the expense of carrying out redundant duplicate computation in parallel.

2) The data distribution algorithm attempts to utilize the aggregate main memory of the system more efficiently.It is a communication-happy algorithm that requires nodes to broadcast their local data to all other nodes.

3) The *Candidate Distribution* algorithm exploits the semantics of the particular problem at hand both to reduce synchronization between the processors and to segment the database based upon the patterns the different transactions support. This algorithm also incorporates load balancing.

Apriori_Algorithm: The first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets. A subsequent pass, say pass *k,* consists of two phases. First, the frequent itemsets *L,* found in the *(k* - 11th pass are used to generate the candidate itemsets *C,* using the Apriori candidate generation procedure described below. Next, the database is scanned and the support of candidates in *C,* is counted. For fast counting, we need to efficiently determine the candidates in *Ck* contained in a given transaction *t.* A hash-tree data structure is used for this purpose.

L, := (frequent 1-itemsets);
k := 2;
while (Lk-l >4) do
begin
//k represents the pass number
C, := New candidates of size *k* generated from *L,-,;*
forall transactions *t* E *D* do
Increment the count of all candidates in *Ck* that
are contained in *t;*
L, := All candidates in C, with minimum support;
k := *k* + 1;
end
Answer := **u k *Lk;***

To see how well the Count distribution algorithm handles larger problem sets when more processors are available, we performed scaleup experiments where we increased the size of the database in direct proportion to the number of nodes in the system.For these experiments, we fixed the size of the multiprocessor at 16 nodes while growing the database from 25 MB per node to 400 MB per node. Count has very good speedup performance. This performance does however begin to fall short of ideal at eight processors.

Demerits- The Data distribution algorithm lost out because of the cost of broadcasting local data from each processor to every other processor. Our results show that even on a high-bandwidth low-latency system such as an SP2, data redistribution is still too costly.

III.FP-GROWTH LIKE PARALLEL ALGORITHMS

In this section,we gave an overview of FP-Growth like parallel algorithm.We have also mentioned about the demerits of this algorithm.

A. Tree Partition based Parallel Frequent Itemsets m ¨ining on Shared Memory Systems[5]

The main idea is to build only one FP-Tree in the memory, partition it into several independent parts and distribute them to different threads. A heuristic algorithm is devised to balance the workload. Our algorithm can not only alleviate the impact of locks during the tree-building stage, but also avoid the overhead that do great harm to the mining stage. We present the experiments on different kinds of datasets and compare the results with other parallel approaches. The results suggest that our approach has great advantage in efficiency, especially on certain kinds of datasets. As the number of processors increases, our parallel algorithm shows good scalability.

Association rule mining searches for interesting relationships among items in a given data set. One of the most famous examples of association rule mining is the market basket problem. However, as for extremely large datasets, the currently proposed frequent pattern mining algorithms still consume too much time.

1) One solution is to design more efficient mining algorithms to reduce the repeated I/O scans as well as to minimize the memory requirement and calculating time. So algorithms like kDCI , FP-Growth , etc, are proposed.

2) Another alternative solution is to parallelize the algorithm.

The present frequent pattern mining algorithms can be divided into two categories: apriori-like algorithms, which are the implementations of the classical apriori algorithm, and the other ones, which are completely different in structure with the classical apriori algorithm. Among both kinds of algorithms, FP-Growth can achieve good efficiency. It's faster than any of the apriori-like algorithms and it only has to scan the whole database twice.
FP-Growth algorithm is based on tree structures.The algorithm can be divided into two steps.

1) Building FP-Tree

Algorithm 1: FP-tree construction

Input: A transaction database DB and a minimum support threshold ξ.
Output: FP-tree, the frequent-pattern tree of DB.
Method: The FP-tree is constructed as follows.

1. Scan the transaction database DB once.Collect F which is the set of all frequent itemsets , and the support of each frequent item. Sort F in support-descending order as FList, the list of frequent items.

2. Create the root of an FP-tree, T , and label it as "null" for each transaction Trans in DB, do the following.

---

2) Mining from the FP-Tree

It's an iterative procedure: each step produces a set of conditional pattern base and then calculated together.The algorithm is shown below:

Algorithm 2: FP-growth: Mining frequent patterns with FP-tree by pattern fragment growth.

Input: A database DB, represented by FP-tree constructed according to Algorithm 1 , and a minimum support threshold ξ.
Output: The complete set of frequent patterns.
Method: Call FP Growth(FP tree, null).
Procedure FP Growth(Tree, α)
begin
/*Mining single prefix−path FP−tree */
if Tree contains a single prefix path **then**
begin
let P be the single prefix−path part **of** Tree;
let Q be the multipath part **with** the top branching node replaced by a null root;
for each combination (denoted as β) **of** the nodes **in** the path P **do**
generate pattern β**U**α **with** support = minimum support **of** nodes **in** β;
let freq pattern **set**(P) be the **set of** patterns so generated;
end
else let Q be Tree;
/* Mining multipath FP−tree */
for each item ai **in** Q **do**
begin
generate pattern β = aiUα **with** support = ai.support;
construct β's conditional pattern−base **and then** β's conditional FP−tree Tree β;
if Tree β = φ(13) **then** call FP−growth(Tree β, β);
let freq pattern **set**(Q) be the **set of** patterns so generated;
end
return (freq pattern **set**(P)Ufreq pattern **set**(Q)U(freq pattern **set**(P)× freq pattern **set**(Q)))
end

Merits:-

Good efficiency is achieved for the tree-partition based parallel algorithm. The tree-partition based parallel algorithm produces no overhead of extra nodes while multi-tree parallel algorithm produces many redundant extra nodes.Different datasets show different characters. Some datasets requires much more I/O operations than pure processing, so the speed-up ratio will not appear to be so good. However, as the datasets become larger, the performance of the tree-partition based parallel algorithm becomes better.

Demerit:-

For extremely large datasets, the currently proposed frequent pattern mining algorithms still consume too much time.

IV. ALGORITHM USING ULTRAMETRIC TREES

In light of the MapReduce programming model [6]-[8], we design a parallel frequent itemsets mining algorithm called FiDoop [9]. The design goal of FiDoop is to build a mechanism that enables automatic parallelization, load balancing, and data distribution for parallel mining of frequent itemsets on large clusters.

FIUT_Algorithm:-

The FIUT (frequent items ultrametric trees) method employs an efficient ultrametric tree structure, known as the FIUtree, to present frequent items for mining frequent itemsets. The proposed FIUT consists of two main phases within two scans of database D. Phase 1 starts by computing the support for all items occurring in the transaction records. Then, a pruning technique is developed to remove all infrequent items, leaving only frequent items to generate the k-itemsets, where the number of frequent items of a transaction is k in a database. Meanwhile, all the frequent 1-itemsets are generated. Phase 2 is the repetitive construction of small ultrametric trees, the actual mining of these trees, and their release. The approach for mining frequent k-itemsets, as proposed in this study, first builds an independent, relative k-FIU-tree for all k-itemsets, where k from M down to 2, and M denotes the maximal value of k among the transactions in the database. Then, each of the trees is mined separately, without generating candidate k-itemsets. The k-FIU-tree is discarded immediately after frequent k-itemsets are mined. At any given time, only one k-FIU-tree is present in the main memory.

MapReduce-Based FiDoop

1. The first phase of FIUT involving two rounds of scanning a database is implemented in the form of two MapReduce jobs. The first MapReduce job is responsible for the first round of scanning to create frequent one itemsets (Algorithm 1).

Algorithm 1 : Parallel Counting: To Generate All Frequent One-Itemsets

---

Input: minsupport, DBi;
Output: 1-itemsets;
1: **function** MAP(key offset, values DBi)
2: //T is the transaction in DBi
3: **for all** T **do**
4: *items* ← split each T;
5: **for all** item in items **do**
6: output(item, 1);
7: **end for**
8: **end for**
9: **end function**
10: **reduce input: (item,1)**
11: **function** REDUCE(key item, values 1)
12: sum=0;
13: **for all** item **do**
14: sum += 1;
15: **end for**
16: output(1-itemset, sum); //item is stored as 1-itemset
17: **if** sum > minsupport **then**
18: *F − list* ← the (1-itemset, sum) //F-list is a CacheFile storing frequent 1-itemsets and their count.
19: **end if**
20: **end function**

2. The second MapReduce job scans the database again to generate *k*-itemsets by removing infrequent items in each transaction (Algorithm 2)

Algorithm 2: Generate k-itemsets: To Generate All *k*-Itemsets by Pruning the Original Database

Input: minsupport, DBi;
Output: k-itemsets;
1: **function** MAP(key offset, values DBi)
2: //T is the transaction in DBi
3: **for all** (T) **do**
4: *items* ← split each T;
5: **for all** (item in items) **do**
6: **if** (item is not frequent) **then**
7: prune the item in the T;
8: **end if**
9: k-itemset ←(k, itemset) /*itemset is the set of f e
uent items
after pruning, whose length is k */
10: output(k-itemset,1);
11: **end for**
12: **end for**
13: **end function**
14: **function** REDUCE(key k-itemset, values 1)
15: sum=0;
16: **for all** (k-itemset) **do**
17: sum += 1;
18: **end for**
19: output(k, k-itemset+sum);//sum is support of this I t
mset
20: **end function**

3. The second phase of FIUT involving the construction of a *k*-FIU tree and the discovery of frequent *k*-itemsets is handled by a third MapReduce job, in which *h*-itemsets (2 <= *h* <= *M*) are directly decomposed into a list of (*h* − 1)-itemsets, (*h* − 2)-itemsets, . . . In the third MapReduce job, the generation of short itemsets is independent to that of long itemsets. In other words, long and short itemsets are created in parallel by our parallel algorithm.

Algorithm 3 : Mining k-itemsets: Mine All Frequent Itemsets

Input: Pair(k, k-itemset+support);//This is the output of the second MapReduce.
Output: frequent k-itemsets;

1: **function** MAP(key k, values k-itemset+support)
2: De-itemset ← values.k-itemset;
3: *decompose*(De-itemset,2,mapresult); /* To d e c
ecompose each De-itemset into t-itemsets (t is f from 2
De-itemset.length), and store the
results to mapresult. */
4: **for all** (mapresult with different item length) **do**
5: //t-itemset is the results decomposed by k-I t e n
mset(i.e. t ≤ k);
6: **for all** (t-itemset) **do**
7: *t − FIU − tree* ← t-FIU-tree generation(local- F I FIU
ee,
t-itemset);
8: output(t, t-FIU-tree);
9: **end for**
10: **end for**
11: **end function**
12: **function** REDUCE(key t, values t-FIU-tree)
13: **for all** (t-FIU-tree) **do**
14: *t − FIU − tree* ← combining all t-FIU-tree from e
ch mapper;
15: **for all** (each leaf with item name v in t-FIU- t r e
tree) **do**
16: **if** (count(v)/| *DB* |≥ minsupport) **then**
17: frequent *h − itemset* ← *pathitem(v)*;
18: **end if**
19: **end for**
20: **end for**
21: **output(h, frequent h-itemset);**
22: **end function**

---

4. The Map function of the third job generates a set of key/value pairs, in which the key is the number of items in an itemset and the value is an FIU-tree that is comprised of nonleaf and leaf nodes. Nonleaf nodes include item-name and node-link; leaf nodes include item-name and its support. In doing so, itemsets with the same number of items are delivered to a single reducer. By parsing the key-value pair (*k*2, *v*2), the reducer is responsible for constructing *k*2-FIU-tree and mining all frequent itemsets only by checking the count value of each leaf in the *k*2-FIU-tree without repeatedly traversing the tree.

V. RELATED WORK

Parallel Mining of Frequent Itemsets:-

Parallel frequent itemsets mining algorithms based on *Apriori* can be classified into two camps, namely, count distribution (e.g., count distribution (CD) [4], fast parallel mining [10], and parallel data mining (PDM) [11]) and data distribution (e.g., data distribution (DD) [4] and intelligent data distribution [12]). In the count distribution camp, each processor of a parallel system calculates the local support counts of all candidate itemsets. Then, all processors compute the total support counts of the candidates by exchanging the local support counts. The CD and PDM algorithms have simple communication patterns, because in every iteration each processor requires only one round of communication. In the data distribution camp, each processor only keeps the support counts of a subset of all candidates. Each processor is responsible for sending its local database partition to all the other processors to compute support counts. In general, DD has higher communication overhead than CD, because shipping transaction data demands more communication bandwidth than sending support counts.

The cascade running mode in existing *Apriori*-based parallel mining algorithms leads to high communication and

synchronization overheads. To reduce time required for scanning databases and exchanging candidate itemsets, FP-growthbased parallel algorithms were proposed as a replacement of the *Apriori*-based parallel algorithms. A few parallel FP-growth-based parallel algorithms (see [5], [13]) were implemented using multithreading on multicore processors. A major disadvantage of these parallel mining algorithms lies in the infeasibility to construct main-memory-based FP trees when databases are very large. This problem becomes pronounced when it comes to massive and multidimensional databases.

VI. CONCLUSION

To solve the performance deterioration, load balancing and scalability challenges of sequential algorithm, various parallel algorithms were implemented. We gave an overview of such parallel algorithms.Unfortunately, in *Apriori*-like parallel FIM algorithms, each processor has to scan a database multiple times and to exchange an excessive number of candidate itemsets with other processors. Therefore, *Apriori*-like parallel FIM solutions suffer potential problems of high I/O and synchronization overhead, which make it strenuous to scale up these parallel algorithms. The scalability problem has been addressed by the implementation of a handful of FP-growth-like parallel FIM algorithms [5]. A major disadvantage of FP-growth like parallel algorithms, however, lies in the infeasibility to construct in-memory FP trees to accommodate large-scale databases. This problem becomes more pronounced when it comes to massive and multidimensional databases.

To solve the challenges in the existing parallel mining algorithms for frequent itemsets, we applied the MapReduce programming model to develop a parallel frequent itemsets mining algorithm called FiDoop. FiDoop incorporates the frequent items ultrametric tree or FIU-tree rather than conventional FP trees, thereby achieving compressed storage and avoiding the necessity to build conditional pattern bases. FiDoop seamlessly integrates three MapReduce jobs to accomplish parallel mining of frequent itemsets. The third MapReduce job plays an important role in parallel mining; its mappers independently decompose itemsets whereas its reducers construct small ultrametric trees to be separately mined.

REFRENCES

[1]M. J. Zaki, "Parallel and distributed association mining: A survey," *IEEE Concurrency*, vol. 7, no. 4, pp. 14–25, Oct./Dec. 1999.

[2]. Pramudiono and M. Kitsuregawa, "FP-tax: Tree structure based generalized association rule mining," in *Proc. 9th ACM SIGMOD Workshop Res. Issues Data Min. Knowl. Disc.*, Paris, France, 2004, pp. 60–63.

[3]M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemsetmining algorithms on MapReduce," in *Proc. 6th Int. Conf. Ubiquit. Inf.Manage. Commun. (ICUIMC)*, Danang, Vietnam, 2012, pp. 76:1–76:8.[Online].Available:http://doi.acm.org/10.1145/2184751.2184842

[4].R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 6, pp. 962–969, Dec. 1996.

[5]D. Chen *et al.*, "Tree partition based parallel frequent pattern miningon shared memory systems," in *Proc. 20th IEEE Int. Parallel Distrib.Process. Symp. (IPDPS)*, Rhodes Island, Greece, 2006, pp. 1–8.

[6]J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008

[7]J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010.

[8]W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," *Proc. VLDB Endow.*, vol. 5,no. 10, pp. 1016–1027, 2012

[9]Y.-J. Tsay, T.-J. Hsu, and J.-R. Yu, "FIUT: A new method for mining frequent itemsets," *Inf. Sci.*, vol. 179, no. 11, pp. 1724–1737, 2009.

[10]D. W. Cheung and Y. Xiao, "Effect of data skewness in parallel mining of association rules," in *Research and Development in Knowledge Discovery and Data Mining*. Berlin, Germany: Springer,1998, pp. 48–60

[11]T. Shintani and M. Kitsuregawa, "Hash based parallel algorithms formining association rules," in *Proc. 4th Int. Conf. Parallel Distrib. Inf. Syst.*, Miami Beach, FL, USA, 1996, pp. 19–30..

[12]E.-H. Han, G. Karypis, and V. Kumar, "Scalable parallel data mining for association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3,pp. 337–352, May/Jun. 2000.

[13]L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor," in *Proc. 33rd Int. Conf. Very Large Data Bases*, Vienna, Austria, 2007, pp. 1275–1285

